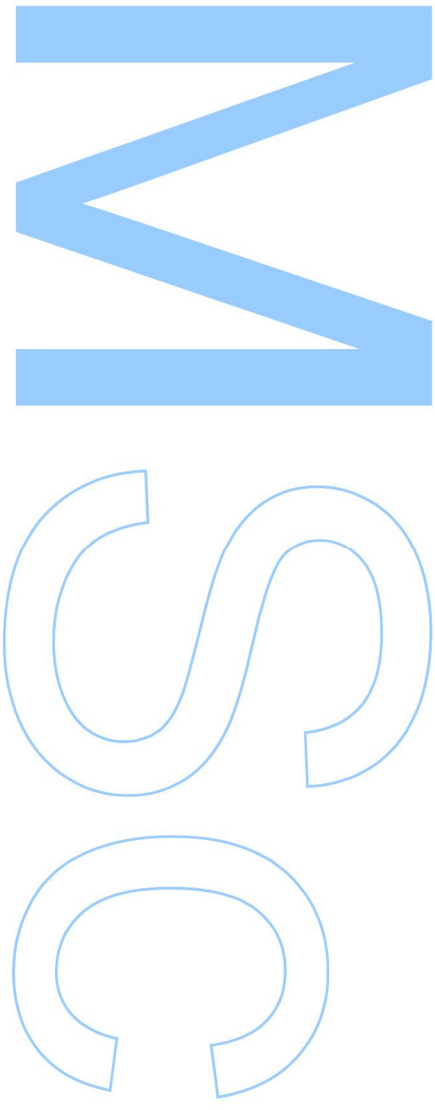
Execução e Gestão de Aplicações Conteinerizadas

Diogo Cristiano dos Santos Reis
Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciências dos Computadores, 2017

Orientador
Professora Inês Dutra, Faculdade de Ciências da Universidade do Porto

Coorientador
Engenheiro Mário Moreira, AlticeLabs



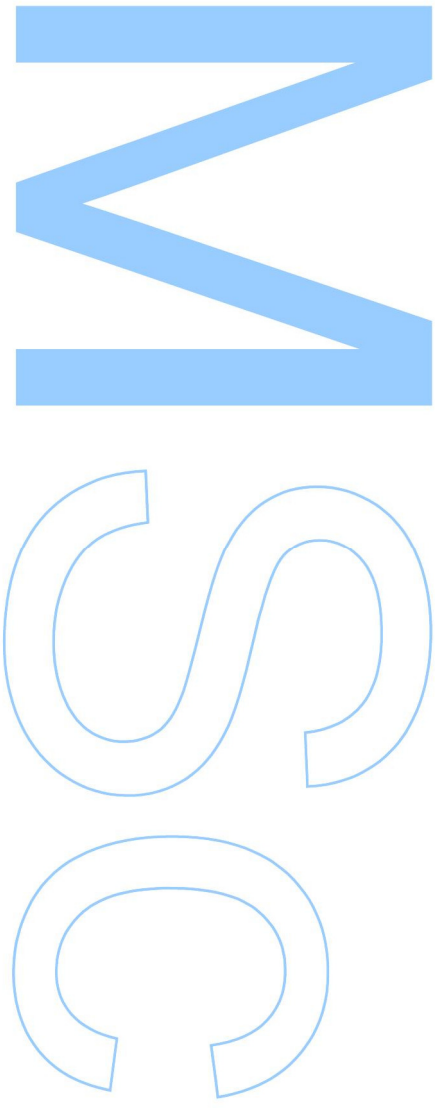
U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____



Abstract

Cloud computing, containerization technologies and containers management have recently gained considerable interest from organizations as a way to accelerate the development and delivery of software to their customers in addition to facilitate the management of such software in the production's environment. This is a critical transformation necessary for the organizations that develop software so they remain competitive in the market. Adopting these new techniques is a complex task that reflects itself in several aspects of the organization, from software development and delivery processes to the necessary physical infrastructure.

This work aims to introduce the applications' contanerization into the software development process of the organizations, identifying the components needed to create a software development pipeline suitable for container concepts. In addition, the Kubernetes platform was evaluated to see if it is capable of performing the deployment of containerized applications as well as managing those applications during its life cycle. In order to perform this evaluation, an application with an architecture based on micro-services was containerized. This application meets a set of requirements capable of evaluating several points of the Kubernetes platform. Another key point on this dissertation is the introduction of elasticity at the node level of the Kubernetes cluster. As the cluster nodes in this work are supported by an OpenStack private cloud, we have manipulated the elasticity of the cloud so that we could automatically launch new nodes when the cluster is at its CPU load limit and thus continue to deploy new containers.

This work leads us to realise that the Kubernetes' plataform is able to carry out the deployment and to manage containers applications in an efficient and agile way. In addition, it demonstrates that Kubernetes is a very complete tool and that besides managing and realizing the deployment of containerized applications in a distributed environment, it has automatic mechanisms that help these applications during their life cycle. By introducing elasticity to the nodes of the Kubernetes cluster it was possible to make it more robust, without the need to manually introduce nodes when there is the necessity to allocate resources to the cluster.

Resumo

A computação na nuvem e as tecnologias de conteinerização e gestão de contentores têm tido nos últimos tempos bastante interesse por parte das organizações como uma forma de acelerar o desenvolvimento e entrega de *software* aos seus clientes e facilitar a gestão desse *software* no ambiente de produção. Esta é uma transformação crítica necessária para as organizações que desenvolvem *software* manterem-se no mercado de uma forma competitiva. Adotar estas novas técnicas é uma tarefa complexa que se reflete em vários aspetos da organização, desde os processos de desenvolvimento e entrega de *software* até à infraestrutura física necessária.

Esta dissertação pretende introduzir a conteinerização de aplicações no processo de desenvolvimento de *software* das organizações, identificando os componentes necessários para criar uma *pipeline* de desenvolvimento de *software* adaptada aos conceitos de contentores. Para além disso, avaliou-se a plataforma Kubernetes para perceber se esta é capaz de realizar o *deployment* de aplicações conteinerizadas e gerir essas aplicações durante o seu ciclo de vida. Para a realização desta avaliação foi conteinerizada uma aplicação com uma arquitetura baseada em micro-serviços que cumpre um conjunto de requisitos capazes de avaliar vários pontos da plataforma Kubernetes. Outro ponto chave desta dissertação é a introdução de elasticidade ao nível dos nós do *cluster* Kubernetes. Como os nós do *cluster* neste trabalho são suportados por um ambiente de *cloud* privada OpenStack foram manipulados os mecanismos de elasticidade da *cloud* para que automaticamente fosse possível lançar novos nós quando o *cluster* está no seu limite de carga de CPU e assim continuar a albergar novos contentores.

Com este trabalho percebeu-se que a plataforma Kubernetes é capaz de realizar o *deployment* e de gerir aplicações conteinerizadas de uma forma eficiente e ágil. Para além disso, percebeu-se que o Kubernetes é uma ferramenta bastante completa e que além de gerir e realizar o *deployment* de aplicações conteinerizadas num ambiente distribuído tem mecanismos automáticos que auxiliam essas aplicações durante o seu ciclo de vida. Com a introdução de elasticidade nos nós do *cluster* Kubernetes foi possível tornar este mais robusto, sem a necessidade de introduzir nós manualmente quando é preciso a alocação de recursos ao *cluster*.

Agradecimentos

No meu percurso académico só consegui ir mais longe por estar rodeado de algumas pessoas que considero gigantes.

Agradeço à professora Inês Dutra pela orientação e dedicação, principalmente no momento da escrita desta dissertação.

Agradeço ao engenheiro Mário Moreira pelos desafios constantes proporcionados que enriqueceram este projeto.

Agradeço a toda a equipa da AlticeLabs, especialmente ao Rui Gonçalves, pela boa integração e pelo apoio na minha primeira experiência num contexto empresarial.

Por fim, mas não menos importante quero agradecer à minha família e aos meus amigos por me terem suportado nos momentos chaves deste meu percurso académico, sem eles nada disto era possível.

A todos vós, muito obrigado.

Conteúdo

Lista de Tabelas

Lista de Figuras

Acrónimos

AMQP *Advanced Message Queuing Protocol*

API *Application Programming Interface*

AWS *Amazon Web Services*

CD *Continuous Delivery*

CERN *Organisation Européenne pour la Recherche Nucléaire*

CI *Continuous Integration*

CLI *Command Line Interface*

CPU *Central Processing Unit*

CRUD *Create*, *Read*, *Update* e *Delete*

DNS *Domain Name System*

GUI *Graphical User Interface*

HOT *Heat Orchestration Template*

IaaS *Infrastructure as a Service*

IoT *Internet of Things*

IP *Internet Protocol*

IPC *Inter-Process Communication*

JDK *Java Development Kit*

JSON *JavaScript Object Notation*

LDAP *Lightweight Directory Access Protocol*

NAT *Network Address Translation*

NFS *Network File System*

NIST *National Institute of Standards and Technology*

PaaS *Platform as a Service*

PAM *Pluggable Authentication Module*

PKI *Public Key Infrastructure*

RPM *Red Hat Package Manager*

SaaS *Software as a Service*

SDK *Software Development Kits*

SDN *Software-Defined Networking*

SFTP *Secure File Transfer Protocol*

SGBD Sistema de Gestão de Bases de Dados

SLA *Service Level Agrement*

SO Sistema Operativo

TAR *Tape Archive*

TI Tecnologia da Informação

TLS *Transport Layer Security*

UFS *Union File System*

USB *Universal Serial Bus*

VPN *Virtual Private Network*

YAML *YAML Ain't Markup Language*

YUM *Yellowdog Updater Modified*

Capítulo 1

Introdução

Com o aparecimento da *web* em meados dos anos 90 desencadeia-se uma nova forma de obter serviços. Os serviços deixam de estar instalados na máquina local e passam para servidores que são acedidos através da rede, mais propriamente a Internet. O que no início eram apenas serviços simples tais como e-mail ou processadores de texto foi-se evoluindo para serviços cada vez mais complexos, sedentos de recursos e suportados por uma infraestrutura de *hardware* gigante. Com o aumento do número de utilizadores com acesso à Internet [1] o cenário foi piorando exigindo cada vez mais dessa infraestrutura. Estas infraestruturas formadas por grandes aglomerados de servidores e equipamentos de rede eram custosos de manter, tinham pouca eficiência, eram difíceis de escalar e de realizar *upgrades*.

Com a introdução dos conceitos de virtualização e da *cloud* as organizações passaram a ter uma infraestrutura ágil no aprovisionamento de máquinas virtuais, mais eficiente na utilização de recursos, com uma maior facilidade na sua operação e com menos custos. No entanto as aplicações não estavam a tirar partido total das capacidades desta nova infraestrutura, pois ainda estavam a ser criadas num formato monolítico. Por esta razão, o processo de desenvolvimento de *software* foi evoluindo para metodologias ágeis e para arquiteturas adaptadas aos conceitos de *cloud*.

Neste novo paradigma *cloud-native*, as aplicações passam a ter uma arquitetura orientada ao serviço. Ou seja, estas são divididas em pequenos módulos ao qual cada um destes módulos representa um serviço dentro da aplicação. A esta arquitetura chamamos micro-serviços. Neste paradigma as aplicações beneficiam ao nível do seu desenvolvimento pois passamos a ter uma aplicação dividida em módulos o que permite mais facilmente criar, gerir e controlar cada um destes módulos. Para além disso, a aplicação passa a ter uma maior flexibilidade em se adaptar a alterações tecnológicas evitando ficar bloqueada em tecnologias com características específicas e é possível de uma maneira mais fácil evoluir a aplicação, acrescentando mais funcionalidades. Outro aspeto benéfico é ao nível da escalabilidade da aplicação. Pois como a aplicação é dividida em pequenos módulos as ações de elasticidade tornam-se mais específicas, escalando apenas os componentes que estão em stress e não a aplicação toda.

Uma tecnologia que vem facilitar e trazer benefícios ao desenvolvimento de aplicações baseadas em *cloud* e em micro-serviços é a utilização de contentores. Contentores são uma tecnologia do sistema operativo Linux, que permite que vários processos possam partilhar o *kernel* do sistema operativo da *host* isoladamente. Ao contrário das máquinas virtuais, esta tecnologia não utiliza virtualização do *hardware* o que aumenta a performance do sistema. Devido a esta razão, os contentores são uma alternativa eficiente à utilização de máquinas virtuais.

Com a chegada da ferramenta Docker [2] a conteinerização de aplicações tornou-se ainda mais simples. Com esta ferramenta os contentores além de serem uma unidade de execução de *software* passam a ser também uma nova forma de empacotamento e distribuição de *software* permitindo que uma aplicação possa ser empacotada, distribuída e executada em qualquer ambiente, desde um computador pessoal até à *cloud*. Como as aplicações são divididas em serviços estas passam a ser compostas por um conjunto de contentores. Gerir e manter um grande número de contentores num ambiente de produção com requisitos de alta disponibilidade e elasticidade requer a utilização de um orquestrador, devido à complexidade inerente a este ambiente. É a este nível que tecnologias como o Kubernetes [3] entram para dar resposta ao *deployment* e gestão de contentores em *clusters* de máquinas físicas ou virtuais.

1.1 Motivação

Como vimos, a introdução da *cloud* e de aplicações conteinerizadas traz benefícios aliciantes às organizações que desenvolvem *software*. Estas organizações encontram-se num ambiente competitivo onde a adoção destes novos paradigmas e tecnologias é essencial para respeitarem os requisitos impostos pelos seus clientes.

Adotar este novo mundo tecnológico para uma organização é um caminho complexo onde apenas algumas empresas conseguiram chegar com sucesso. A adoção destas novas tecnologias implica alterações que se refletem em vários aspetos da organização, desde a infraestrutura física, aos modelos de desenvolvimento de *software* até à mudança de mentalidades nas pessoas envolvidas. Esta complexidade torna difícil a entrada neste novo mundo tecnológico o que leva a que muitas organizações não tomem as melhores decisões para o caminho a seguir, fazendo com que o processo de transformação seja mais lento, com mais custos e perdas. É com base na complexidade da adoção destas novas tecnologias que motivou a realização desta dissertação.

1.2 Objetivos

O objetivo principal desta dissertação é propor um caminho que permita a introdução das tecnologias em torno dos contentores numa organização, para que esta seja capaz de criar uma plataforma sobre um ambiente *cloud* que permita o desenvolvimento, gestão e execução de aplicações conteinerizadas de uma forma moderna e escalável.

Para que este objetivo possa ser alcançado, sub-objetivos foram definidos, que são:

- Identificar as vantagens e desvantagens que a conteinerização de aplicações pode trazer a uma organização que desenvolve *software*;
- Descrever os desafios e os problemas existentes no momento da adoção deste novo paradigma, assim como as soluções para esses problemas;
- Determinar quais as tecnologias e mecanismos necessários para desenvolver aplicações conteinerizadas de forma eficiente sobre uma *pipeline* de desenvolvimento de *software* que utiliza práticas de integração e entrega contínua (CI/CD) de *software*;
- Criar um *cluster* Kubernetes em alta disponibilidade e elástico que permita a gestão e *deployment* de contentores;
- Introduzir elasticidade na plataforma Kubernetes ao nível dos nós do *cluster*, de maneira a que seja possível a introdução de novos nós, com base na carga de CPU, sem intervenção manual. A elasticidade deve ser introduzida ao nível da infraestrutura (IaaS) e manipulada através da interface de linha de comandos pcloud;
- Criar uma prova de conceito que exponha num cenário prático a conteinerização de uma aplicação numa arquitetura de micro-serviços composta tanto por serviços sem estado assim como serviços com estado, demonstrando as várias fazes do processo de conteinerização assim como o seu *deployment*;
- Realizar uma avaliação do cenário prático anterior que demonstre que o conjunto de tecnologias adotadas e as metodologias criadas são capazes de suportar uma aplicação durante o seu ciclo de vida completo, desde a fase de desenvolvimento até à fase de entrada em produção.

O estudo e as implementações criadas no âmbito desta dissertação vão ser realizadas na AlticeLabs [4], que é uma organização que há mais de 65 anos desenvolve e fornece serviços digitais de comunicação em Portugal. Quando esta dissertação foi iniciada, esta empresa estava a passar por um processo de modernização tecnológico ao nível da sua *cloud* privada, que na altura apenas suportava o conceito de IaaS e estava a dar os primeiros passos ao nível das plataformas de *cloud* para o desenvolvimento e produção de *software*. Devido a isto, os objetivos a que esta dissertação se propõe adequasse com pertinência ao momento em que a organização se encontra, daí ser o local adotado à realização desta dissertação.

1.3 Organização

No presente Capítulo 1 foi realizada uma introdução para contextualizar o leitor sobre o paradigma e os desafios que implicam a adoção de tecnologias de gestão e conteinerização de aplicações nas organizações. Logo de seguida foi apresentada a motivação que levou a traçar os objetivos que esta dissertação se propõe a concretizar, a partir deste ponto esta dissertação está organizada da seguinte forma:

No Capítulo 2 são abordados os conceitos fundamentais relacionados com os sistemas de

computação na *cloud*, as técnicas que permitem a virtualização e a conteinerização de aplicações, as arquiteturas num estilo de micro-serviços, as estratégias existentes que permitem fornecer a uma plataforma elasticidade e por fim são abordadas as práticas automáticas de desenvolvimento de *software*.

No Capítulo 3 são abordadas com algum detalhe as tecnologias e ferramentas que foram selecionadas para esta dissertação. Estas são o OpenStack que permite a criação de *clouds* privadas, as interfaces via linha de comandos que permitem interagir com a *cloud*, a plataforma Docker que permite a conteinerização de aplicações e por fim é abordado a ferramenta de gestão e *deployment* de aplicações conteinerizadas chamada Kubernetes.

O Capítulo 4 é essencialmente a parte de desenvolvimento desta dissertação, onde é mostrado os elementos necessários para criar uma *pipeline* de desenvolvimento de *software* conteinerizado assim como os fatores chaves que uma organização deve ter em conta para adoção destas novas tecnologias. Após isso é apresentado as formas de instalação de um *cluster* Kubernetes expondo os elementos mínimos necessários para obter um *cluster* deste género. Logo de seguida, são apresentadas as implementações realizadas para fornecer elasticidade horizontal ao *cluter* Kubernetes através de uma linha de comandos. Por fim, são apresentadas as funcionalidades do Kubernetes que lidam com algumas das necessidades das aplicações durante o seu ciclo de vida. Estas funcionalidades estão para além das funcionalidades de gestão e *deployment* de contentores.

No Capítulo 5 é apresentada uma aplicação que respeita um conjunto de requisitos que vai permitir a sua conteinerização e sucessivo *deployment* sobre um *cluster* Kubernetes de maneira a avaliar as abordagens sugeridas nesta dissertação.

No Capítulo 6 serão discutidos os resultados obtidos anteriormente onde serão abordadas, com base nesses resultados, as vantagens e desvantagens que a conteinerização de aplicações e a utilização das tecnologias Docker e Kubernetes podem trazer a uma organização.

No Capítulo 7 é feita uma reflexão sobre todo o trabalho realizado e onde serão apresentados pontos desta dissertação a desenvolver no futuro.

Capítulo 2

Fundamentos

Este capítulo fornece uma explicação dos conceitos que serão abordados no decorrer desta dissertação. Vamos começar por apresentar a definição, características e modelos que fazem parte do conceito de *cloud* para depois disso podermos aprofundar os conceitos inerentes à virtualização comparando-os com o conceito de contentores. Após isso, vamos abordar a conteinerização de aplicações que possibilitam a adoção de arquiteturas em micro-serviços, para que logo de seguida sejam introduzidos os conceitos de elasticidade nestas aplicações e na *cloud*. Por fim, serão abordados as estratégias modernas e automáticas para o desenvolvimento de *software*.

Com a introdução destes conceitos espera-se reunir o conhecimento suficiente que vão permitir mais facilmente entender os aspetos desenvolvidos no âmbito desta dissertação.

2.1 Cloud Computing

O termo "nuvem" associado à computação foi introduzido quando as operadoras de telecomunicações começaram a disponibilizar serviços de *Virtual Private Network* (VPN) para a comunicação de dados [5]. Para além das VPN, o conceito de computação na nuvem evolui de paradigmas como computação em *cluster* e computação em *grid*, chegando aos dias de hoje como uma forma de abstração dos recursos computacionais e da infraestrutura física de um *data center* composta por servidores, *hardware* de armazenamento e rede [6]. Esta camada de abstração possibilita um acesso simples aos recursos computacionais, fornecendo esses recursos como se uma espécie de *self-service* em catálogo se tratasse [7]. Com isto as equipas de desenvolvimento ganham bastante poder na gestão e aprovisionamento da infraestrutura que suportam as suas aplicações.

Este conceito, promete reduzir custos, fornecer flexibilidade e escalabilidade o que fez com que as empresas das TIs adiram rapidamente à *cloud*.

Nesta dissertação, como a computação na nuvem é o conceito que abrange todos os outros conceitos e é a base deste trabalho, decidiu-se que seria o ponto de partida. Sendo assim, nesta secção vamos abordar os conhecimentos necessários sobre a definição, modelos e estrutura do

conceito de computação na nuvem.

2.1.1 Definição e Características

A definição de computação na nuvem continua a ser algo que divide bastantes opiniões, pois este conceito normalmente é definido entre os utilizadores à custa da utilização que cada um emprega. De todas as definições existentes e criadas aquela que tem sido mais utilizada e aceite pelos investigadores deste tema é a que foi criada e publicada pelo *National Institute of Standards and Technology* (NIST) [7]. Esta organização criou a definição de computação na nuvem tendo em vista as características essenciais, os tipos de serviços fornecidos pela *cloud* e os modelos de implementação utilizados.

Como características essenciais a NIST definiu que os serviços fornecidos tinham de ser ***On-demand Self-service***, ou seja, o utilizador tem a possibilidade de aprovisionamento de capacidades da *cloud* (por exemplo: servidores ou armazenamento) de forma automática e simples, sem a necessidade de intervenção de terceiros. Estes serviços fornecidos ao cliente devem ser de ***Rapid Elasticity***. Isto é, o cliente a qualquer momento pode, de maneira automática, requisitar mais ou menos recursos consoante as suas necessidades. Os recursos são organizados em grupos (***Resource Pooling***) e é utilizado um modelo de múltiplos inquilinos, aos quais, estes recursos são alocados conforme as necessidades de cada um. A localização dos utilizadores deve ser independente da localização da *cloud* e esses utilizadores não devem ter qualquer tipo de controlo ou conhecimento exato da sua localização dentro do *data center*, que compõe a *cloud*. O conhecimento deve ficar apenas por regiões geográficas, como por exemplo país ou continente.

Os serviços e as aptidões da *cloud* são fornecidos pela rede e o utilizador deve aceder através dos meios comuns existentes (***Broad Network Access***). Estes serviços consumidos pelos clientes devem ser medidos de forma rigorosa e deve existir um modelo justo de cobrança e pagamento implementado (***Measured Service***).

2.1.2 Modelos de Serviço

A organização NIST definiu três modelos de serviço que agrupam as diferentes utilizações da infraestrutura da nuvem:

- ***Infrastructure as a Service* (IaaS):** Os serviços de infraestrutura são tipicamente recursos de computação, armazenamento, e rede que o cliente tem disponíveis de forma virtual e automática. Esta infraestrutura normalmente é utilizada para a instalação de máquinas virtuais com sistemas operativos que correm aplicações. O utilizador, neste modelo de serviço, apenas tem controlo das suas máquinas virtuais e redes montadas na estrutura da *cloud*. A administração da estrutura subjacente apenas diz respeito ao provedor de serviço. Por exemplo, o Amazon EC2 [8] ou Microsoft Azure [9] são consideradas soluções para serviços de IaaS.

- ***Platform as a Service* (PaaS):** PaaS é um serviço de computação que consiste numa plataforma de alojamento e implementação de *software* que disponibiliza ferramentas para que os utilizadores (por exemplo: programadores) possam desenvolver e customizar as suas aplicações de uma forma coordenada, rápida e automatizada. Graças ao PaaS as aplicações de desenvolvimento, ferramentas de compilação, bibliotecas, bases de dados, servidores aplicacionais, e entre outros, não têm de ser instalados nas máquinas desses utilizadores. Estes elementos são instalados centralizadamente na *cloud* e fornecidos na forma de plataformas, prontas a ser utilizadas. Para além disso, os utilizadores não necessitam de criar ou controlar a estrutura técnica dessas plataformas, tais como, rede, sistema operativo, e armazenamento. Por exemplo, o Google App Engine [10] e o Heroku [11] são exemplos de provedores de PaaS.
- ***Software as a Service* (SaaS):** O SaaS é uma forma de distribuição de *software*, no qual o provedor é responsável por toda a estrutura necessária para a disponibilização da aplicação aos clientes. O cliente apenas é responsável por adquirir, utilizar e customizar o *software*. Os serviços são acedidos por uma interface disponibilizada através da Internet, normalmente um *browser* ou uma API. Por exemplo, o *software* Google Docs [12] e Gmail [13] são fornecidos num modelo de serviço SaaS.

2.1.3 Modelos de Implementação

Uma vez que nem todas as organizações desejam que todos os utilizadores possam aceder e utilizar certos recursos no seu espaço de computação na nuvem ou então requerem um sistema próprio que possa ser adaptado mais especificamente às suas exigências e os dados guardados por vezes são considerados sensíveis demais para serem armazenados num local público. Foram introduzidos quatro tipos de implementações diferentes, que são:

- ***Cloud* Privada:** É um tipo de *cloud* que é utilizado por uma única entidade, normalmente uma empresa ou organização. A infraestrutura física utilizada pertence à organização ou a uma terceira parte que a mantém, existindo um controlo e operação exclusivo e total por parte da organização, em relação às instâncias que são executadas. Este tipo de infraestrutura normalmente é utilizado por um grupo reduzido de pessoas, por exemplo pelos funcionários e clientes dessa organização. Esta solução oferece uma maior segurança e transparência, mas requer um investimento significativo na aquisição de toda a infraestrutura envolvente. Os sistemas OpenStack [14] ou CloudStack [15] são exemplos de tecnologias capazes de criar um ambiente de *cloud* privada.
- ***Cloud* Comunitária:** A infraestrutura da nuvem pertence e é partilhada por diversas organizações sendo destinada a uma comunidade específica que partilha as mesmas preocupações, características ou interesses. Esta pode ser controlada por apenas uma ou várias organizações da comunidade, ou ainda uma terceira parte.
- ***Cloud* Pública:** Este é o modelo de implementação de *cloud* mais conhecido e utilizado. A infraestrutura, neste tipo de *cloud*, pertence ao provedor e os serviços estão disponíveis através de uma rede pública (normalmente a Internet) a múltiplos utilizadores. Frequen-

temente estão associados a este tipo de nuvem custos baixos e uma infraestrutura com possibilidade de ser bastante escalável. Por exemplo, o Amazon EC2 [8] ou Microsoft Azure [9] são consideradas nuvens públicas.

- ***Cloud* Híbrida:** É a composição dos modelos públicos, privados ou comunitários, falados anteriormente. Neste tipo de nuvem é permitido que uma nuvem privada, quando está no limiar da sua utilização de recursos, veja os seus recursos aumentados usando para isso os recursos de uma nuvem pública.

2.2 Virtualização e Contentores

Virtualização é uma técnica que permite abstrair plataformas de *hardware*, sistemas operativos, dispositivos de armazenamento e de rede, criando uma versão virtual de vários tipos de recursos computacionais. A virtualização desempenha um papel determinante na *cloud* pois maior parte dos serviços que fornece, como armazenamento ou serviços de computação, só são possíveis empregando os conceitos de virtualização.

Nesta secção vamos abordar o conceito de *hypervisor* e o seu propósito para perceber a sua importância na virtualização ao nível do *hardware*. Após isso será abordado os conceitos que estão por traz dos contentores para depois comparar esta tecnologia em relação às máquinas virtuais. Com isto, será possível que nas próximas secções se entenda mais facilmente conceitos mais complexos e infimamente ligados à containerização e gestão de contentores.

2.2.1 *Hypervisors*

Entre o *hardware* e as máquinas virtuais existe a camada que fornece a virtualização, que é conhecida como *hypervisor*. Este sistema tem o objetivo de controlar as plataformas virtuais e gerir os seus sistemas operativos. Ou seja, o *hypervisor* fornece os recursos de *hardware* de uma forma padronizada abstraindo esses recursos e fornecendo a ilusão às máquinas virtuais de que estas estão a correr diretamente na máquina física [16].

Os *hypervisors* são organizados em dois tipos básicos: tipo 1, por vezes, denominado de *bare metal hypervisor* e o tipo 2, por vezes, denominado de *hosted hypervisor* [17]. Na Figura 2.1 é possível visualizar a disposição dos *hypervisors* em relação ao *hardware* e às máquinas virtuais.

***Hypervisor* Tipo 1:**

Este tipo de *hypervisor* é instalado diretamente sobre o *hardware*. Isto permite que depois do *hypervisor* tipo 1 estar instalado e configurado as instâncias sejam montadas diretamente sobre ele e os sistemas operativos correm apenas dentro das máquinas virtuais. Xen ou VMware ESX/ESXi são exemplos deste tipo de *hypervisor*.

***Hypervisor* Tipo 2:**

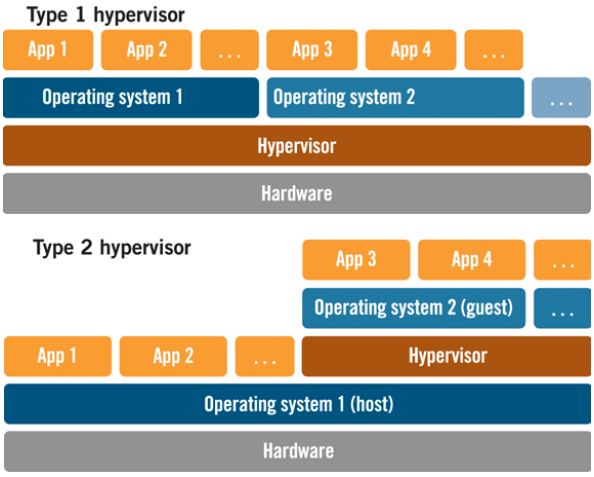


Figura 2.1: Comparação dos *hypervisors* tipo 1 e tipo 2. (Figura extraída de [18])

O *hypervisor* do tipo 2 funciona como um programa normal que corre sobre um sistema operativo comum, como por exemplo Linux ou Windows. Isto torna possível que dentro deste programa se consiga correr máquinas virtuais com outro sistema operativo. KVM e VirtualBox são exemplos deste tipo de *hypervisor*.

Tal como visto em [19], os *hypervisors* de tipo 1 têm melhor eficiência do que os de tipo 2, isto deve-se ao facto dos *hypervisors* de tipo 1 correrem diretamente sob o *hardware* o que reduz a sobrecarga e torna o sistema mais rápido. Esta é a razão pelo qual os *hypervisors* de tipo 1 normalmente são a melhor solução para serem implementados na *cloud*.

2.2.2 Virtualização ao Nível do Hardware (Máquinas Virtuais)

Virtualização ao nível do *hardware* consiste em virtualizar o *hardware* através de um *hypervisor* e em cima deste adicionar máquinas virtuais que fornecem uma abstração, tal e qual, como se de uma máquina física se tratasse. Depois disto, dentro de cada máquina virtual é executado um sistema operativo completo que é independente e isolado, suportando as diferentes aplicações.

Neste nível os *hypervisors* suportam diferentes estratégias para gerir os recursos da máquina física e partilhá-los pelas máquinas virtuais. O *hypervisor* também tem a responsabilidade de criar o isolamento entre as máquinas virtuais que suporta[20].

Na *cloud* este tipo de virtualização baseada em *hypervisors* é ideal quando as aplicações que são utilizadas requerem diferentes sistemas operativos, como por exemplo Linux ou Windows [21].

2.2.3 Contentores

Com a utilização de contentores não existe qualquer tipo de virtualização, pois contentores não são virtualização [22]. Os processos a correr dentro dos contentores interagem diretamente com o *kernel* Linux da *host*. Cada um destes contentores encapsula um grupo de processos que são isolados dos outros processos a correr nos outros contentores, fornecendo ao processo a ilusão que estão a ser executados numa máquina isolada e dedicada.

Os contentores são uma alternativa leve à virtualização, pois cada contentor não tem sistema operativo. Isto faz com que seja possível correr centenas de contentores numa única máquina, ao contrário do que acontecia com os *hypervisors* e as máquinas virtuais.

Os contentores assentes no sistema operativo Linux, utilizam três mecanismos fundamentais do Linux para fornecer o isolamento e a conteinerização de aplicações:

- **Cgroups:** é a abreviatura de *control groups* e é um mecanismo do *kernel* do Linux que é capaz de controlar e limitar a alocação dos recursos (CPU, memória, I/O e rede) a grupos de processos desse sistema [23].
- **Namespaces:** é outro mecanismo do *kernel* do Linux que fornece abstrações aos recursos globais do *kernel* e faz com que os processos tenham a ilusão que têm uma instância isolada nesse recurso global [24]. Esses recursos podem ser IDs de processos, IDs de utilizadores, *hostnames*, comunicação entre processos (IPC), rede e sistemas de ficheiros [22].
- **Chroot:** permite alterar o diretório raiz de um processo, de tal maneira que o processo não consegue aceder a ficheiros fora deste novo diretório [25].

2.2.4 Comparação entre Máquinas Virtuais e Contentores

Como vimos anteriormente, a tecnologia de virtualização empregue nas máquinas virtuais difere da dos contentores, o que faz com que, isso crie diferenças no desempenho, manobrabilidade e nos casos de uso que cada uma das plataformas pode ter.

A Figura 2.2 ilustra a comparação ao nível da arquitetura entre máquinas virtuais e contentores. A principal diferença visível é que cada máquina virtual contém um sistema operativo completo e todos os binários e bibliotecas necessárias para a execução da aplicação, levando a uma sobrecarga na utilização dos recursos da *host*. Pelo contrário, os contentores partilham isoladamente o sistema operativo da *host* e por isso não incluem dentro destes outro sistema operativo completo, apenas incluem os binários e bibliotecas para execução da aplicação.

Na tabela 2.1 é apresentada outra comparação entre máquinas virtuais e contentores, percorrendo vários parâmetros chaves comuns às duas plataformas.

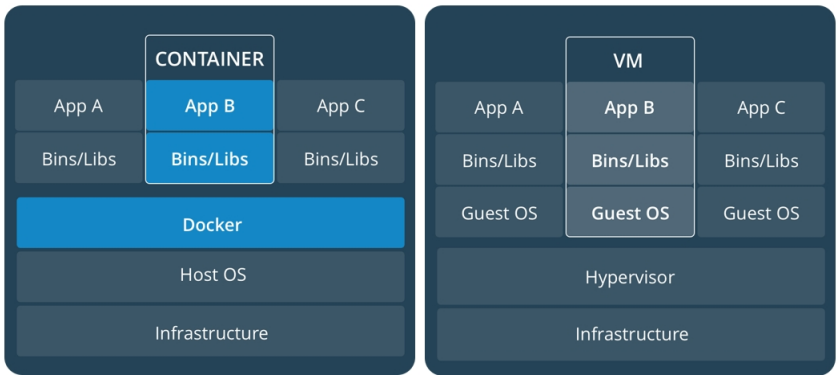


Figura 2.2: Comparação de contentores e máquinas virtuais ao nível da sua arquitetura. (Figura extraída de [26])

Tabela 2.1: Comparação entre a virtualização baseada em máquinas virtuais e em contentores.

Parâmetro	Máquinas Virtuais	Contentores	Fonte
Performance	Performance Inferior	Performance próxima do *bare-metal*	[27, 28]
Segurança	Superfície de Ataque Inferior	Superfície de Ataque Superior	[29]
Isolamento	Isolamento Superior	Isolamento Inferior	[30]
Memória	Utilização Superior	Utilização Inferior	[31]
Armazenamento	Utilizam um ou mais discos virtuais para fornecer espaço de armazenamento.	Utilizam um conceito de dados sem estado.	[32]
Tempo de Inicialização	Minutos	Segundos	[33]
Diversidade de SOs suportados	Suportam vários tipos de kernels de SOs	Apenas suportam o *kernel* do SO da *host*	[31]

2.3 Arquitetura de Micro-serviços

Micro-serviços é um estilo de uma arquitetura de desenvolvimento de aplicações em que esta é composta por múltiplos serviços independentes no qual cada um destes serviços corresponde a um processo que comunicam uns com os outros através de mecanismos de comunicação simples, como por exemplo uma API HTTP [34]. Estes serviços podem ser implementados com tecnologias e linguagens de programação diferentes, utilizar modelos de dados próprios e a sua gestão é realizada a partir de um ponto central, por um serviço separado.

Em oposição à abordagem de micro-serviços encontramos as aplicações desenvolvidas num estilo monolítico que são compostas por uma única unidade de grandes dimensões. Estas aplicações podem ser compostas na mesma por vários serviços, mas o seu *deployment* é realizado apenas por um elemento. Esta abordagem traz dificuldades na realização de alterações, pois a cada alteração realizada na aplicação requer que seja criada uma nova versão de toda a aplicação e seja feito um *deployment* total. Outra dificuldade encontrada na utilização de arquiteturas monolíticas é relativa a escalabilidade. Sempre que é necessário realizar ações de escalabilidade em algum componente temos de escalar a aplicação completa levando a um desperdício de recursos. Estas aplicações normalmente são compostas por uma grande quantidade de código com um grau de complexidade alto o que torna difícil gerir e adicionar novos elementos à equipa de desenvolvimento durante o ciclo de vida da aplicação [35].

Todos os problemas associados à utilização de uma estrutura monolítica no desenvolvimento de aplicações é eliminada se utilizarmos uma arquitetura dividida em micro-serviços. Devido a isto, nos últimos anos a popularidade deste estilo de arquitetura aumentou bastante nas organizações. Um estudo desenvolvido pela Nginx conclui que 70% dos 1,825 inquiridos estavam a utilizar ou a investigar o conceito de micro-serviços e que quase 1/3 utiliza este tipo de arquitetura em produção [36].

2.4 Elasticidade

Sendo a *cloud* uma estrutura ágil capaz de albergar múltiplas aplicações com baixos custos, fez com que se tornasse num local aliciante aos produtores de *software* que colocam lá os seus produtos e que contam com a *cloud* como uma forma de lidar com um grande número de pedidos a que estão sujeitos. Estas aplicações suportadas por máquinas virtuais ou contentores para aproveitar ao máximo as capacidades da *cloud* devem ser capazes de se redimensionar automaticamente de forma a lidar com um número de pedidos flutuante, a cada momento. Estes pedidos não são constantes e por vezes têm picos que ultrapassam a média, daí a necessidade de um mecanismo de elasticidade automático. Este mecanismo elástico deve ser capaz de escalar os recursos da infraestrutura subjacente das aplicações de maneira a que estas lidem com um grande número de pedidos ou então em alturas em que o tráfego baixa estes recursos possam ser dispensados, tornando o sistema eficiente.

Nesta secção vamos abordar e definir o conceito de elasticidade na *cloud*, explorando as diferentes estratégias que implementam essa elasticidade.

2.4.1 Definição de Elasticidade

Segundo Herbst et al. [37], elasticidade é o ponto em que um sistema é capaz de se adaptar às mudanças da carga de trabalho por prover ou desprover recursos de forma automática, de tal forma, que em cada momento os recursos disponibilizados correspondem à demanda atual o mais próximo possível das necessidades.

2.4.2 Classificação das Soluções de Elasticidade

Como proposto por Galante and de Bona [38], as soluções existentes para alcançar elasticidade podem ser agrupadas em diferentes classes no que diz respeito, ao seu local, política, propósito e método, tal como ilustrado na Figura 2.3. O resto desta secção é dedicado a este tema.

Local

A primeira característica define o local onde as ações de elasticidade são controladas. Estas podem ser controladas através do sistema operativo da infraestrutura da *cloud* (no nível IaaS),

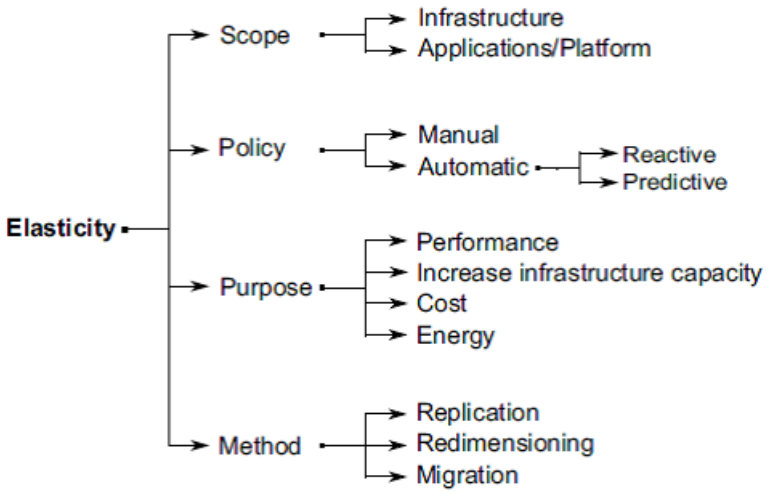


Figura 2.3: Classificação das Soluções Elásticas (Figura extraída de [38])

na própria aplicação (no nível SaaS) ou na plataforma da aplicação (no nível PaaS), ou seja pode ser controlada e implementada em qualquer uma das camadas da *cloud*. Frequentemente, vemos a elasticidade implementada ao nível de IaaS, onde o controlador da *cloud* fornece os recursos às máquinas virtuais de acordo com métricas delas. A elasticidade também pode ser implementada no nível da PaaS ou SaaS. Neste caso, o controlador é incorporado num destes níveis e interage com o sistema operativo da *cloud* de maneira a alocar ou a libertar os recursos necessários.

Política

Quanto à política das soluções de elasticidade, estas podem ser consideradas manuais ou automáticas. Uma solução elástica **manual** fornece as ferramentas necessárias para alocar ou desalocar recursos, mas o utilizador é responsável por monitorizar, aplicar e executar todas as ações de elasticidade. Esta política não é considerada elasticidade pois de acordo com a definição apresentada anteriormente estes mecanismos devem ser automáticos.

Ao contrário da política manual, na política **automática** é o próprio sistema que toma as decisões de elasticidade de acordo com algumas regras definidas pelo utilizador, normalmente estas regras constam do *Service Level Agrement* (SLA). Para que o sistema seja capaz de tomar decisões à cerca do momento em que deve escalar, este tem de ter em tempo real informações e dados (métricas) à cerca da utilização dos recursos, como por exemplo utilização de CPU, memória ou rede. O sistema pode desencadear as ações que fornecem elasticidade de forma **preditiva**. Ou seja, as ações são desencadeadas tendo em conta o histórico de carga do sistema ou então padrões de carga que levam a despoletar as ações de elasticidade. Para que isto seja possível, é utilizado heurísticas, técnicas matemáticas e analíticas que antecipam o comportamento da carga do sistema. Outra forma conhecida de desencadear as ações de elasticidade e aquela que é mais comum é chamada de **reativa**. Estas soluções são baseadas em mecanismos de Regra-Condição-Ação, em que uma regra é composta por um conjunto de condições que quando satisfeitas desencadeiam um conjunto de ações que fornecem a elasticidade.

Propósito

A elasticidade é uma das características chaves dos sistemas da *cloud* e pode ter vários propósitos. A elasticidade fornece eficiência, pois a cada momento apenas se utiliza a quantidade necessária de recursos para o funcionamento do sistema. Isto, traz vantagens tanto para o provedor de serviços como para o utilizador, pois reduz a utilização de recursos e consequentemente os custos. Outro propósito e bastante notório ao nível do utilizador é que a performance das aplicações albergadas na *cloud* mantêm-se constante, pois mesmo que o tráfego aumente os recursos serão aumentados de forma automática de maneira a lidar com esse aumento de tráfego. Outro propósito vantajoso é que a elasticidade ao utilizar e gerir de forma eficiente os recursos do sistema da *cloud* faz com que se utilize menos energia.

Método

A última característica tem a ver com os métodos utilizados para implementar a elasticidade nos sistemas. Existe três maneiras fundamentais de fornecer e implementar elasticidade nas aplicações que correm em grupos de instâncias. Esta pode ser implementada por escalabilidade horizontal, escalabilidade vertical e escalabilidade por migração [39].

Escalabilidade horizontal consiste em replicar, ou seja, é quando aumentamos ou diminuímos o número de instâncias virtuais alocadas a um serviço. Estas instâncias podem ser máquinas virtuais, contentores ou módulos (no caso de SaaS). Neste tipo de escalabilidade por replicação é necessário um componente extra capaz de rotear e dividir os pedidos pelas várias réplicas das instâncias, habitualmente utiliza-se um balanceador de carga. Uma vantagem de utilizar este tipo de escalabilidade é que se torna a aplicação mais robusta a falhas, porque introduz redundância no sistema e caso uma das instâncias falhe existe outras a correr o mesmo serviço capazes de continuarem a lidar com os pedidos dos utilizadores.

Na **escalabilidade vertical** normalmente apenas existe uma única máquina virtual e a elasticidade é conseguida ao redimensionar a cota de capacidades dessa única máquina, tais como CPU, memória ou rede. A principal vantagem deste método, é que se consegue adaptar o sistema mais facilmente e rapidamente à demanda de pedidos, pois ao contrário da escalabilidade horizontal não é necessário criar uma nova máquina com todos os componentes instalados para correr o serviço, basta aumentar ou diminuir a cota de recursos. Esta forma de elasticidade apenas é exequível se for possível interagir com o *hypervisor* da *cloud* com bastante autoridade, o que faz com que se torne uma opção mais complexa e insegura. Para além disso, os sistemas operativos comuns não suportam alterações nos seus componentes chaves, como CPU e memória durante o tempo de execução.

Por fim, é introduzido o último método de **elasticidade por migração**, ao qual este método tem implicações ao nível físico. Neste método a elasticidade é implementada com a migração da máquina virtual para outra máquina física que melhor enquadra a carga a que a máquina virtual está sujeita no momento.

2.5 Práticas de Desenvolvimento de *Software*

Nesta secção vamos abordar as práticas de desenvolvimento de *software* com o objetivo de perceber as diferentes etapas que um *software* deve passar até chegar ao ambiente de produção.

2.5.1 Introdução

As empresas das TI que ainda não adotaram metodologias de desenvolvimento de *software* modernas e ágeis realizam a sua produção em ciclos bastante extensos, que podem demorar semanas ou até meses. Para além disso, qualquer alteração ao produto após o início de desenvolvimento torna-se numa tarefa bastante complexa e por vezes impossível. Estas empresas utilizam um modelo de produção em cascata, como o da Figura 2.4. Neste modelo o produto de *software* só passa para a fase seguinte quando a fase em que se encontra estiver totalmente concluída. Depois do *software* estar totalmente desenvolvido é necessário a realização de testes o que requer outra grande quantidade de tempo, pois maior parte dos testes são feitos com métodos manuais e pouco automáticos.

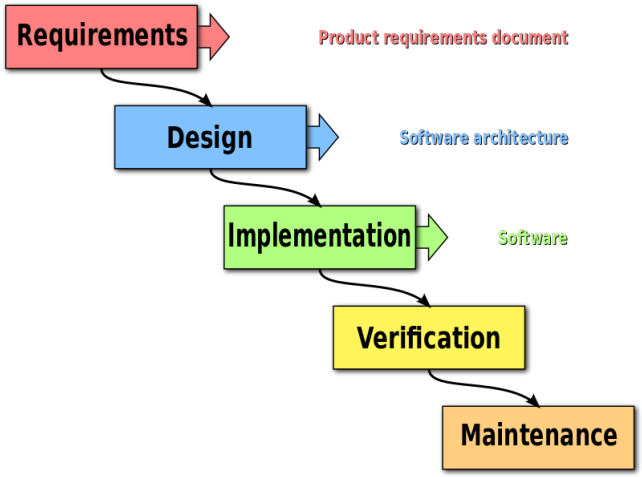


Figura 2.4: Modelo de Desenvolvimento de *Software* em Cascata (Figura extraída de [40])

É evidente que para reduzirmos os ciclos de desenvolvimento de *software* é necessário introduzir metodologias automáticas e repetitivas. A automatização deve ser introduzida em todas as fazes tais como compilação, testes e *deployment*. Com isto, o tempo demorado no processo global de desenvolvimento de *software* é reduzido [41].

A introdução da *cloud* no mundo das empresas das TIs lado a lado com as novas metodologias de desenvolvimento de *software*, fez com que revoluciona-se a maneira como os *softwares* são desenvolvidos. Pois, permite que a automatização seja implementada nas *pipelines* de desenvolvimento mais facilmente, aumentando a frequência de lançamentos e reduzindo o tempo de desenvolvimento dos *softwares* [42].

Antes de aprofundarmos os conceitos sobre os métodos e práticas automáticas de produção

de *software*, vamos primeiro abordar o conceito de ambiente de desenvolvimento e ambiente de produção, pois é sobre estes ambientes que os conceitos apresentados nas próximas secções vão ser exercidos.

2.5.2 Ambiente de Desenvolvimento e Ambiente de Produção

O ambiente de desenvolvimento é o local onde os programadores desenvolvem e criam os produtos de *software* ou acrescentam novas funcionalidades. É também neste ambiente que são realizados todos os testes pelas equipas de controlo de qualidade. Assim que o *software* tenha passado todos os testes, este é colocado no ambiente de produção, em que existe as equipas de operação. Estas equipas têm o objetivo de manter e assegurar que o *software* chega ao cliente com todos os requisitos que este definiu. Ou seja, o ambiente de produção é local onde se vai realizar o *deployment* do *software* criado que está em contacto com o cliente.

2.5.3 *Continuous Integration* (CI)

Continuous Integration é um método de desenvolvimento de *software* em que os programadores de uma equipa de desenvolvimento criam, cada um deles, uma parte do código do produto e frequentemente (no mínimo uma vez por dia) vão realizando *commits* para um repositório de código de maneira a que este código seja integrado frequentemente com o restante código num servidor de integração contínua. A este processo de integração está associado um conjunto de testes que automaticamente detetam incompatibilidades e erros no processo de integração fornecendo *feedback* ao programador de maneira a que os problemas sejam corrigidos. Após os erros serem corrigidos é iniciado um novo processo de integração desse código.

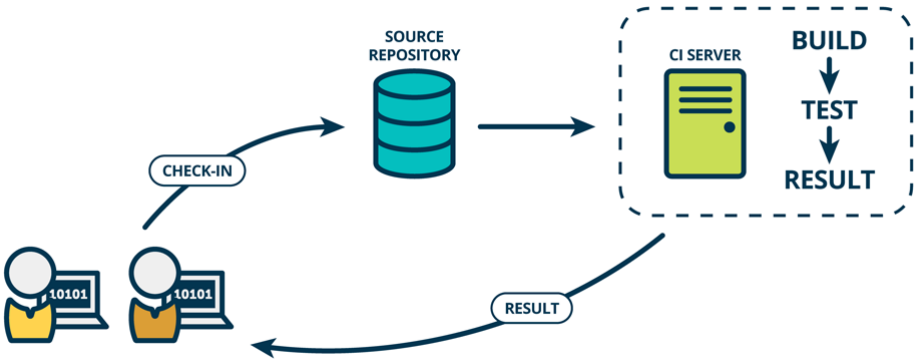


Figura 2.5: Ciclo do método *Continuous Integration* (Figura extraída de [43])

Existe um grupo de ferramentas que devem existir num ambiente de desenvolvimento de *software* que utiliza da melhor maneira possível o método de CI [44]. Como se pode ver pela Figura 2.5 uma destas ferramentas é repositório central de código fonte. Devido à complexidade e ao grande número de ficheiros que cada produto de *software* pode conter, as equipas de

desenvolvimento passaram a adotar ferramentas de gestão e versionamento de código fonte que armazenam todos os ficheiros que compõem o produto (exemplos: *scripts* de teste, ficheiros para o *deployment*, esquemas de bases de dados, ficheiros de configuração, etc), assim como as suas diferentes versões ao longo do tempo. Para que as equipas de desenvolvimento tirem o melhor partido possível de um sistema como este é aconselhado que o repositório seja central e partilhado por toda a equipa de desenvolvimento. Git e Subversion [45] são bons exemplos de ferramentas com o propósito de gestão e versionamento de código fonte.

Outro componente que é necessário para atingir o método de CI e que pode ser visto na Figura 2.5 é o servidor de integração contínua. Este servidor é responsável por detetar alterações ao código no repositório central e sempre que essas alterações existam este inicia um novo processo de integração desse código. Um exemplo de um *software* que é capaz de criar este servidor de integração contínua é o Jenkins [46].

Jenkins é uma plataforma *open source* de CI escrita em Java. Esta plataforma é bastante popular devido à sua capacidade de adaptação a qualquer ambiente de desenvolvimento, com centenas de *plugins* que permitem aumentar a extensibilidade desta plataforma [47]. Estes *plugins* são capazes de suportar sistemas de controlo de versão, ferramentas de *build*, métricas de qualidade de código, integração com sistemas externos, customização da interface gráfica, entre outros.

Todos os testes realizados neste método são ao nível da integração das diferentes partes que compõe o produto final. Mesmo depois de passar estes testes de integração, normalmente, o produto ainda não pode ser enviado para produção. Pois ainda é necessário realizar um estágio com testes que simulam um ambiente muito idêntico ao de produção, que serão abordados na próxima secção, no método de *Continuous Delivery* (CD).

2.5.4 *Continuous Delivery* (CD)

Continuous Delivery é uma metodologia de desenvolvimento e entrega de *software* onde se mantém o produto, durante todo processo de desenvolvimento, num estado permanente de lançamento para produção. Por outras palavras, isto significa que quando os programadores realizam alterações ao código este vai ser integrado no servidor de integração e depois de passar os testes de integração o código é submetido a uma *pipeline* com testes mais avançados. Esta *pipeline* recria um ambiente de estágio muito parecido com o ambiente de produção, fornecendo em todas as fases *feedback* às equipas de desenvolvimento, de maneira a realizarem correções e otimizações. Como os testes realizados, logo após a integração, têm um ambiente muito idêntico ao de produção, faz com que o *software* se encontre sempre pronto para lançamento [48].

Como se pode ver pela Figura 2.6, um dos componentes chaves deste método é a existência de uma plataforma que permita criar *pipelines* de testes. O tipo e o número de etapas que compõe a *pipeline* pode diferir consoante o *software* que as equipas estão a desenvolver. Na Figura 2.7 é possível ver um exemplo genérico de uma *pipeline* de *Continuous Delivery* em que inclui os

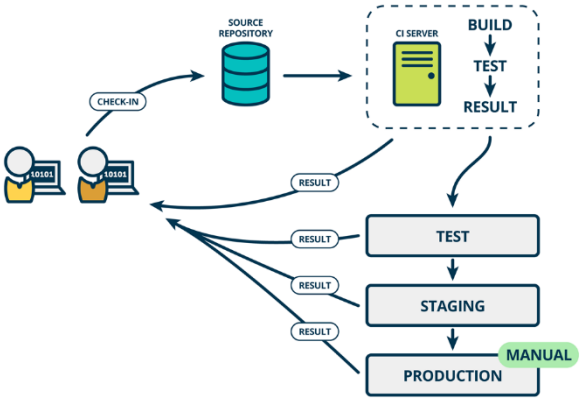


Figura 2.6: Ciclo do método *Continuous Delivery* (Figura extraída de [43])

vários testes que vão atestar se as alterações ao *software* estão aptas a entrar em produção.

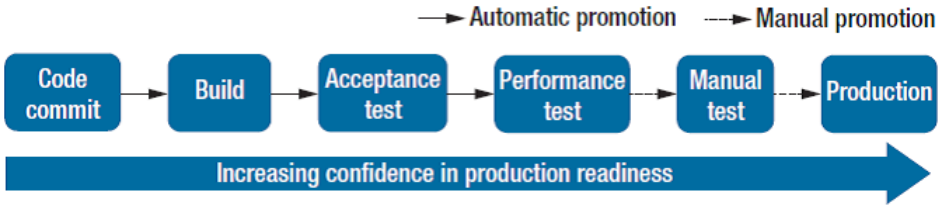


Figura 2.7: Exemplo de uma *pipeline* de *Continuous Delivery* (Figura extraída de [48])

Segundo Chen [48] as *pipelines* de CD são normalmente compostas pelos seguintes estágios:

Code Commit

Normalmente este é o estágio inicial da *pipeline* e é iniciado automaticamente quando o programador envia o código para o repositório central. Neste estágio o código submetido vai ser compilado e são realizados testes unitários. Caso seja detetado algum erro, o programador é informado de maneira a que possa corrigir o erro e enviar novamente o código corrigido para a *pipeline*. Se não existir nenhum erro, o código passa automaticamente para o próximo estágio, que é o *Build*.

Build

Nesta fase o código é integrado e são realizados os testes de integração, de maneira a verificar se existe ou não alguma incompatibilidade. Caso exista algum problema o *software* não avança na *pipeline* para que os programadores possam corrigir o erro. Também é nesta fase que as diferentes partes do código e as suas dependências são integradas e transformadas num género de artefacto ou objeto (por exemplo RPM ou imagem de um contentor), ao qual as próximas etapas, da *pipeline*, vão empregar esse objeto criado. Tal como no estágio anterior se não for detetado nenhum problema o *software* avança para a próxima fase da *pipeline*.

Acceptance Test

Esta é uma parte crucial que determina o sucesso do método CD, pois vai assegurar que o *software* produzido contém todas as especificações e requisitos exigidos pelos clientes. Normalmente, estes testes tem um conceito de caixa negra pois são fornecidos um conjunto de *inputs* ao sistema (de acordo com cenários reais) e consoante esses *inputs* são devolvidos *outputs*. Estes *outputs* são analisados de maneira a verificar se estão de acordo com aquilo que era esperado. Caso todos os testes sejam positivos passamos automaticamente à próxima fase da *pipeline*.

Performance Test

Nesta fase da *pipeline* vamos testar como ficou a performance do *software*, depois do programador ter realizado as alterações ou melhorias ao código. Ou seja, os programadores após um *commit* ficam a saber instantaneamente as alterações que provocaram no *software*, ao nível da performance. Com isto, mais facilmente poderão realizar alterações de maneira a corrigir a performance global do produto.

Manual Test

Esta é a fase antes do *software* ser lançado para produção. É nesta fase que são realizados todos os testes manuais ao produto.

Production

Esta é a ultima fase da *pipeline* de CD e é nesta fase que o *software* passa de um possível produto a produto, ou seja, é nesta fase que se encontra o utilizador final (cliente).

2.5.5 *Continuous Deployment*

Esta prática de produção de *software* é muito semelhante à prática de *Continuous Delivery*. A diferença aqui está no nível de automatização, pois todo o *software* produzido pelas equipas de desenvolvimento entra automaticamente para produção depois de totalmente testado na *pipeline*. Ou seja, o último passo da pipeline do *Continuous Delivery* é realizado automaticamente e não manualmente, como se pode ver na Figura 2.8.

Para que este método tenha sucesso a *pipeline* de realização de testes automáticos tem de ser bastante eficaz. Pois caso contrário, poderia chegar ao ambiente de produção *software* com erros e falhas, colocando em risco tanto a qualidade do produto, assim como o interesse do cliente por esse produto. Devido a isto, os testes automáticos devem cobrir toda as fases desde os testes unitários, testes de componentes e testes de aceitação [49].

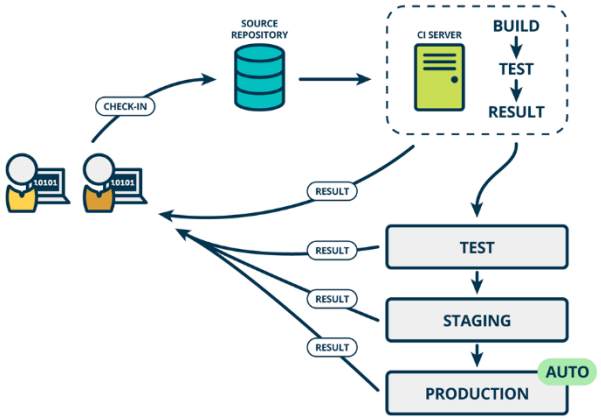


Figura 2.8: Ciclo do método *Continuous Deployment* (Figura extraída de [43])

Capítulo 3

Estado da Arte

Este capítulo fornece uma visão das tecnologias e produtos de *software*, que são atualmente o estado da arte para implementar os objetivos que esta dissertação se propõe a concretizar. Para isso será divido em três secções, ao qual, cada uma destas secções vai reunir as tecnologias e ferramentas que são capazes de atingir cada uma das soluções a implementar, referidas anteriormente na secção 1.2.

Inicialmente vamos abordar o *software* utilizado para a criação de *clouds* privadas com o foco nos seus mecanismos de elasticidade para reunirmos conhecimento sobre este sistema de maneira a introduzir a elasticidade nos nós do *cluster* Kubernetes. Como a elasticidade que será introduzida nos nós do *cluster* será manipulada por uma CLI, serão abordadas as CLI de duas das *clouds* públicas mais populares e a CLI pcloud. Depois vamos abordar a plataforma de conteinerização Docker e por fim vamos abordar as plataformas que são capazes de realizar a gestão e *deployment* de contentores.

3.1 OpenStack

Os sistemas de controlo e operação em *cloud* são responsáveis por um grande número de tarefas o que faz com que estes sistemas sejam bastante complexos. As suas funções vão desde o fornecimento dos serviços da *cloud* ao controlo e coordenação do sistema. Estes sistemas normalmente tem uma estrutura modular, em que, a cada um dos módulos correspondem os sistemas de computação, de rede, de armazenamento, de gestão de imagens, de identidades e de segurança. Depois de instalados, estes sistemas fornecem a camada de IaaS e os utilizadores interagem com eles através de uma interface que pode ser gráfica (GUI), por uma linha de comandos (CLI), ou por uma API. Existem várias versões deste tipo de sistema, tanto comerciais como *open-source*, cada uma delas tem os seus objetivos e desafios. A utilização de cada uma destas opções, depende do ambiente, ou as necessidades das organizações intervenientes.

Como já foi referido anteriormente na secção 1.2, esta dissertação está a ser elaborada na Altice Labs, na qual, já tinha uma plataforma de IaaS privada implementada através do *software*

open-source OpenStack. Esta *cloud* privada será a infraestrutura subjacente ao *cluster* Kubernetes que para além disso vai permitir fornecer elasticidade horizontal aos nós desse *cluster*. Para isso será necessário perceber os sistemas internos que compõe o OpenStack para que se consiga interagir com estes para criar essa elasticidade.

De seguida vamos abordar a arquitetura do sistema OpenStack e a forma como os serviços são disponibilizados aos seus utilizadores.

3.1.1 Introdução

O OpenStack [14] é um sistema operativo para *cloud*, de código aberto, que permite gerir e controlar vários recursos computacionais disponíveis num *data center*, mais propriamente computação, armazenamento e rede de forma virtual, criando um serviço de infraestrutura (IaaS) privada.

O projeto OpenStack começou em 2010 através do trabalho e colaboração entre a Rackspace Hosting e a NASA. Estas duas organizações decidiram juntar dois projetos desenvolvidos de forma independente. A Rackspace forneceu o CloudFiles que é uma plataforma de armazenamento para a *cloud*. Enquanto que a NASA disponibilizou o Nova, criado no projeto Nebula, em que as suas funções eram ao nível da computação na *cloud*. Desta forma nasce um projeto de código aberto apoiado e desenvolvido pela OpenStack Foundation que incentiva e organiza a participação de qualquer pessoa no projeto. Para além dos voluntários, ao redor do mundo, que desenvolvem e aprimoram o OpenStack, este também conta com o apoio de mais de 200 companhias [50], no qual estão presentes os gigantes desta indústria, como por exemplo IBM, Intel, Red Hat, SUSE, Canonical, Cisco, entre muitos outros. [51]

Todo o *software* do OpenStack está sob a licença Apache 2.0 e encontra-se neste momento num ciclo de lançamento de 6 meses, ao qual, a última versão é denominada de Ocata e foi lançada em Fevereiro de 2017.

3.1.2 Arquitetura

O OpenStack fornece uma solução de *Infrastructure as a Service* (IaaS) através de um *design* modular. Cada um destes módulos é chamados de Serviço ou Projeto OpenStack e apenas está presente consoante os serviços e funcionalidades que cada *cloud* fornece aos seus utilizadores. Cada um destes serviços tem o objetivo de gerir e manipular diferentes recursos da estrutura física da *cloud*, que são abstraídos e virtualizados para serem fornecidos ao utilizador. A organização e disposição destes componentes está relacionada com quatro conceitos fundamentais do projeto OpenStack, que são:

- **Computação:** Serviço Nova;
- **Armazenamento:** Serviço Swift e Serviço Cinder;
- **Rede:** Serviço Neutron;

- **Acesso:**
  - **Identidades:** Serviço Keystone;
  - **Imagens:** Serviço Glance;

Estes serviços são compostos por diferentes processos *daemons*, ao qual, estes comunicam entre si através de uma fila central e de um protocolo avançado de enfileiramento de mensagens (Broker AMQP). A cada serviço está associado uma base de dados que tem a função de guardar os diferentes estados de alguns dos componentes da *cloud*, como por exemplo o estado das máquinas virtuais a correr no sistema. Para além disto, a cada um destes serviços inter-relacionados está associada uma API, ou noutros casos um *proxy* que serve de interface de comunicação entre os diferentes projetos do OpenStack. A fila central de mensagens, a base de dados e a API formam uma arquitetura genérica presente em todos os serviços do OpenStack, como pode ser visto pela Figura 3.1. Os utilizadores interagem com o OpenStack de várias formas, tais como, interface gráfica (Dashbord Horizon), linha de comandos (CLI) e também através da API.

Para além dos serviços essenciais supracitados, também referidos como serviços *core* do OpenStack, é de salientar que existem outros que são conhecidos como serviços opcionais. Tendo identificado os principais serviços/projetos que compõem a arquitetura lógica do OpenStack, de seguida vamos abordar com mais detalhe cada um destes serviços que compõe o OpenStack e fazer algumas referências aos serviços opcionais que podem ser instalados também.

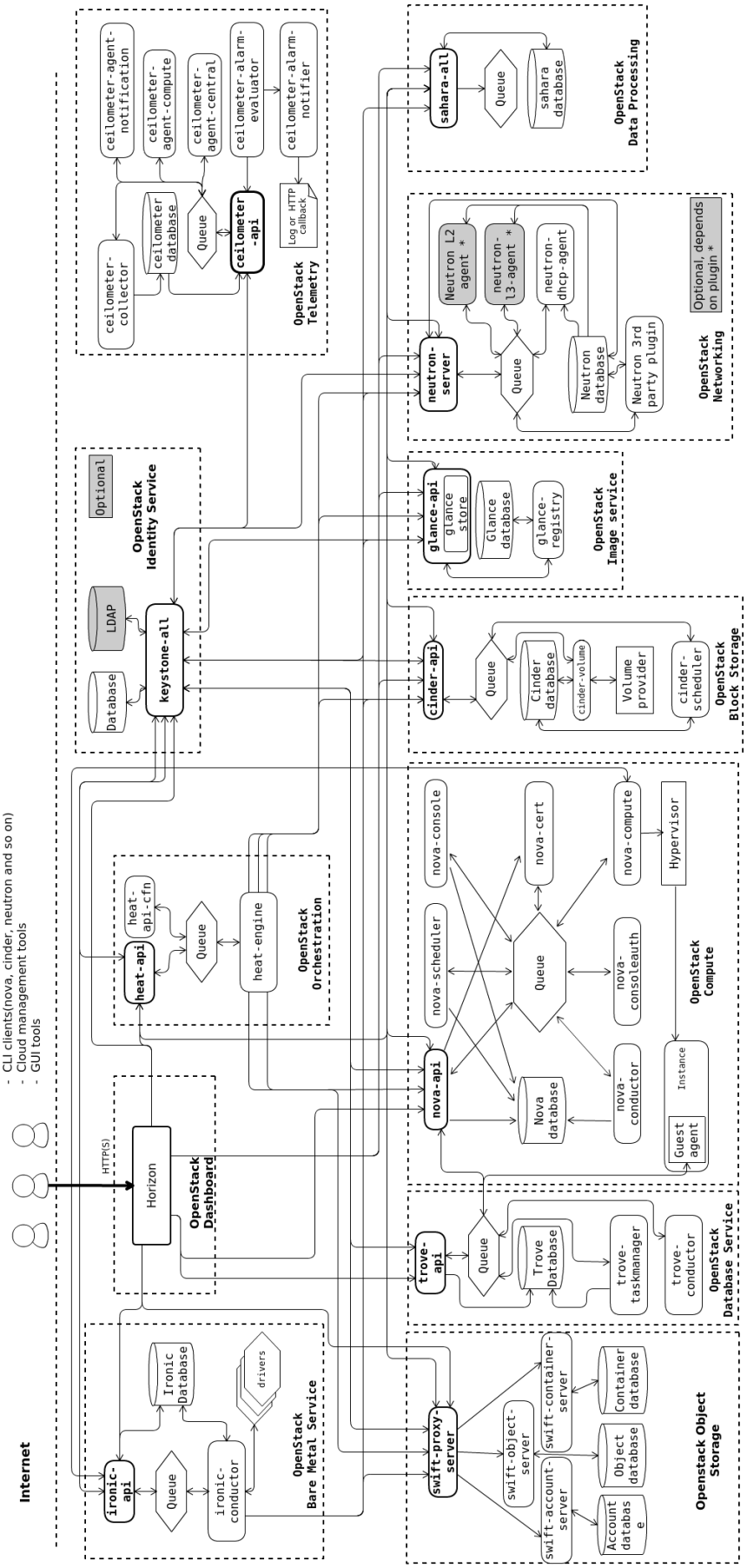


Figura 3.1: Arquitetura mais comum do OpenStack (Figura sob a licença Apache 2.0 extraída de [14])

3.1.3 Serviços Essenciais

De seguida vamos abordas os serviços essenciais e aqueles que são considerados mais importantes no OpenStack.

3.1.3.1 Nova (Compute)

O projeto Nova já está presente desde os inícios do projeto OpenStack e é o componente responsável pela gestão das instâncias (máquinas virtuais). Este é um sistema distribuído por múltiplos servidores, ao qual, é responsável por criar uma camada de abstração dos recursos de computação desses servidores. Este componente comunica diretamente com o *hypervisor* fazendo com que seja possível gerir o ciclo de vida de uma instância num qualquer número de nós físicos do *data center*. Desta forma, as principais funcionalidades deste componente passam por iniciar, redimensionar, suspender, parar e reiniciar máquinas virtuais.

As instâncias geridas pelo Nova tem os seus recursos distribuídos e catalogados em Flavors [52]. Um Flavor define a capacidade de CPU, memória e armazenamento alocado a uma máquina virtual.

3.1.3.2 Neutron (Networking)

O projeto Neutron disponibiliza um serviço de *Networking-as-a-Service* através da abordagem de *Software-Defined Networking* (SDN), ao qual, permite criar e gerir redes, *subnets* e portas pelos utilizadores do OpenStack.

Tal e qual como Nova, este segue uma abordagem modular, de tal forma, que pode ser colocado num servidor dedicado a este serviço e os seus agentes e extensões distribuídos pelos diferentes componentes de rede que compõem a infraestrutura da *cloud*. Desta forma, permite acomodar diferentes equipamentos e *softwares* de rede trazendo mais flexibilidade à *cloud*. O serviço de rede no OpenStack fornece uma API que permite gerir e configurar uma grande variedade de serviços/dispositivos de rede, tais como, *switches*, *routers*, *firewalls*, NAT, VPN, balanceamento de carga, entre outros.

3.1.3.3 Glance (Image)

Este é o serviço responsável por fornecer aos utilizadores as imagens externas dos sistemas operativos a correr nas máquinas virtuais. A maneira como o Glance disponibiliza as imagens é semelhante à de um sistema operativo contido num *live* CD.

As imagens fornecidas aos utilizadores são compostas por um sistema de ficheiros e um sistema operativo em que todos os meta-dados, específicos ao *host*, são removidos. Os elementos removidos podem ser, chaves SSH, endereços MAC, endereços IP estáticos, entre outros. Estas

especificidades são eliminadas, porque caso contrário quando vários utilizadores colocassem estas imagens a correr iriam existir conflitos, por exemplo devidos às duplicações de endereços.

3.1.3.4 Swift (Object Storage)

Swift é o componente do OpenStack responsável por gerir o armazenamento de objetos. Estes objetos são considerados dados estáticos que são armazenados a longo prazo e podem ser atualizados ou recuperados a qualquer momento.

Este sistema de armazenamento de objetos utiliza uma arquitetura distribuída sem um ponto central de controlo, fornecendo desta forma, grande escalabilidade, redundância e permanência dos objetos armazenados. Os objetos são escritos e replicados pelo próprio serviço Swift por múltiplos nós de armazenamento, que compõe o *cluster*.

3.1.3.5 Cinder (Block Storage)

Cinder é outra forma de armazenamento do OpenStack, este é responsável por gerir e fornecer o acesso a dispositivos de armazenamento físico, em blocos, às instâncias.

Como as imagens disponíveis pelo serviço Glance não têm nenhuma forma de armazenamento persistente, no momento em que uma instância fosse desligada perder-se-ia toda a informação criada até esse momento. É este o problema que o Cinder vem resolver. Pois este componente é responsável por criar volumes lógicos que depois de anexados a uma instância fornece armazenamento persistente a essa instância.

Para além de fornecer armazenamento às instâncias, este também tem outras funcionalidades tais como [53]:

- Gestão de *Snapshots*: permite criar/eliminar *snapshots* de volumes.
- Permite clonar volumes.
- Permite a criação de volumes tendo por base um *snapshot* de um outro volume.
- Permite copiar imagens para volumes, ou vice-versa.

De maneira a melhorar a performance e a velocidade com que cada instância realiza operações de escrita/leitura, a maioria do *hardware* de armazenamento permite que as instâncias comuniquem diretamente com estes.

Os recursos básicos fornecidos pelo Cinder são:

- **Volumes:** armazenamento em blocos alocado de maneira a fornecer armazenamento secundário ou primário às instâncias para que possam ser iniciadas (*boot*).
- ***Snapshots*:** é uma cópia de um volume, num determinado momento de atividade desse volume. Mais tarde o *snapshot* pode ser convertido novamente num volume, tal e qual como quando foi criado o *snapshot*.

- ***Backups*:** são cópias de volumes que são armazenados como objetos no Swift.

3.1.3.6 Keystone (Identity)

Este é o serviço responsável pela gestão de identidades do OpenStack. Como todos os serviços do OpenStack exigem mecanismos de autenticação e de identidade, o Keystone é considerado o serviço que une todos os outros serviços. Este serviço agrega as funções de autenticação, listagem do catálogo de serviços disponíveis, políticas, regras e credenciais que permitem controlar os acessos dos diferentes utilizadores associados a cada projeto. Os pedidos realizados aos diferentes serviços do OpenStack através das APIs são processados pelo Keystone de forma assegurar que o utilizador certo pode utilizar o serviço requisitado.

O Keystone suporta vários *plugins*, *backends* e protocolos para auxiliar o processo de autenticação e identificação de identidades como o LDAP e o PAM. Para além destes, utiliza o mecanismo tradicional de utilizador/*password* de maneira a requisitar uma *token* e para os restantes pedidos utiliza uma *Public Key Infrastructure* (PKI), sendo este o método *default* de autenticação do OpenStack [53].

3.1.4 Serviços Opcionais

Neste momento, existem 13 serviços opcionais que podem ser instalados na infraestrutura do OpenStack, que são: Horizon, Ceilometer, Heat, Trove, Sahara, Ironic, Zaqar, Manila, Designate, Barbican, Magnum, Murano e Congress. Estes ao serem instalados estendem as capacidades que o OpenStack pode fornecer aos seus utilizadores. De todos os serviços opcionais, apenas nos vamos focar naqueles que podem contribuir para esta dissertação. Esses serviços são: o Heat (Orquestração) devido aos seus mecanismos de escalabilidade automática e o Ceilometer (Telemetria) devido à recolha de métricas que permitem despoletar as ações de elasticidade.

3.1.4.1 Heat (Orchestration)

Heat é o módulo OpenStack responsável pela orquestração. Orquestração é o processo de lançar vários recursos (por exemplo máquinas virtuais) que estão interligados de maneira a cooperarem e trabalharem em conjunto. Isto é feito à custa de um ficheiro, denominado *template*, que permite definir quais os recursos a serem provisionados [54], que vão ser falados em mais detalhe no final desta secção.

Para além do conceito de *template*, outro conceito importante no Heat é o conceito de *stack* (pilha). Um *template* quando é lançado o que está na realidade a fazer é criar uma *stack* de recursos. Estes recursos podem ser, instâncias, volumes, *routers* virtuais, *firewalls*, balanceadores, entre outros. Como a tarefa do Heat é orquestrar e não configurar, este para além de interligar os recursos e os tornar disponíveis, também controla o momento em que determinados recursos vão iniciar e tornar-se ativos.

Para além disso, também é responsável por passar os dados iniciais de configuração. Imediatamente após o *boot* da instância, o Heat realizará algumas tarefas simples, que normalmente passa por chamar motores de execução *pos-boot* mais complexos que iram desempenhar funções mais especificas e relacionadas com a configuração dos sistemas.

Templates

Como referido anteriormente os recursos que são manipulados pelo módulo Heat são especificados através de ficheiros chamados *templates*. Estes seguem o formato *Heat Orchestration Template* (HOT) e são escritos na linguagem YAML [55]. Seguem uma estrutura organizada em sete secções, como pode ser visto a seguir.

```
heat_template_version:
  # versão do template (necessário)
description:
  # descrição do template (opcional)
parameter_groups:
  # declaração da organização e ordem dos inputs (opcional)
parameters:
  # declaração dos parâmetros dados por inputs (opcional)
resources:
  # declaração dos recursos a provisionar (necessário)
outputs:
  # declaração dos outputs (opcional)
conditions:
  # declaração das condições (opcional)
```

De seguida vão ser explicadas cada uma das secções em mais detalhe de maneira a reunir conhecimento na escrita de *templates* HOT. No anexo A está disponível um *template* que será usado na secção 4.6.3 desta dissertação e serve de exemplo completo da escrita de um *template*.

Secção `heat_template_version`:

Os *templates* HOT são sempre iniciados pelo campo versão, que é a data das várias *releases* do projeto Heat. O conteúdo deste campo permite identificar qual o formato do *template* e as funcionalidades suportadas nessa versão. Até à data existem oito versões suportadas que são: 2013-05-23, 2014-10-16, 2015-04-30, 2015-10-15, 2016-04-08, 2016-10-14, 2017-02-24, 2017-09-01.

Secção `description`:

Como o próprio nome indica, esta secção serve para indicar o propósito do *template*.

Secção `parameter_groups`:

Esta secção é opcional e permite especificar e organizar como é que os parâmetros de *input* são associados e agrupados tendo em conta os recursos e a ordem como devem ser fornecidos. Por exemplo podemos agrupar os parâmetros `srv_flavour` e `srv_image` como sendo os parâmetros de um servidor e os parâmetros `db_flavour` e `db_image` como sendo de um sistema de bases de dados.

Secção `parameters`:

Esta secção define os parâmetros de *input* a serem introduzidos no momento de lançamento do *template*. Estes parâmetros permitem tornar o *template* mais customizado e não específico a um conjunto de características. Por exemplo permite definir qual a imagem de uma instância a ser utilizada no momento do *deployment* do *template*.

Os parâmetros são organizados em blocos individuais e iniciados pelo nome do parâmetro. Cada parâmetro tem um conjunto de campos que permitem especificar e regular o parâmetro em questão, como se pode ver a seguir:

```
parameters:
  <param name>:
    type: <string | number | json | comma_delimited_list | boolean>
    label: <human-readable name of the parameter>
    description: <description of the parameter>
    default: <default value for parameter>
    hidden: <true | false>
    constraints:
      <parameter constraints>
    immutable: <true | false>
```

No decorrer do *template* um parâmetro de *input* pode ser referenciado através da função `get_param`.

Secção `resources`:

Esta secção define e especifica os recursos que vão compor a *stack* a ser orquestrada pelo módulo Heat. Estes recursos podem ser instâncias, volumes, redes, etc. Cada recurso é definido num bloco individual como no caso anterior dos `parameters`, como se pode ver pelo seguinte excerto:

```
resources:
  <resource ID>:
    type: <resource type>
    properties:
      <property name>: <property value>
    metadata:
      <resource specific metadata>
    depends_on: <resource ID or list of ID>
    update_policy: <update policy>
    deletion_policy: <deletion policy>
    external_id: <external resource ID>
    condition: <condition name or expression or boolean>
```

Cada bloco `resources` tem um conjunto de campos que são:

- **`<resource ID>`:** define o identificador/nome do recurso que é atribuído pelo utilizador (por exemplo servidor, cluster, redes);

- **type:** define o tipo do recurso (por exemplo `OS::Nova::Server` ou `OS::Neutron::Port`). Uma lista de todos recursos disponíveis pode ser vista em [56];
- **properties:** permite especificar as propriedades do recurso. Por exemplo no caso de ser um *server* permite definir a imagem, o *flavour*, etc;
- **metadata:** define meta-dados que serão associados aos recursos.
- **depends_on:** permite definir dependências entre recursos. Por exemplo se o servidor X depender do servidor Y, o servidor X só será iniciado depois do Y.
- **update_policy:** permite definir restrições as atualizações dos recursos.
- **deletion_policy:** permite definir o que acontece ao recurso quando este é eliminado. As opções disponíveis são *Delete*, *Retain* e *Snapshot*.
- **external_id:** permite especificar o identificador de um recurso externo.
- **condition:** permite definir condições que com base nos parâmetros de *input* decidem se o recurso deve ser criado ou não.

Secção `outputs`:

A secção `outputs` define a informação que deve ser mostrada após a *stack* ser criada. Por exemplo pode expor o número de instâncias criadas, os endereços IP's dessas instâncias, etc.

Esta secção é criada em blocos como podemos ver a seguir:

```
outputs:
  <parameter name>:
    description: <description>
    value: <parameter value>
    condition: <condition name or expression or boolean>
```

Cada um destes blocos de `outputs` tem um conjunto de parâmetros:

- **<parameter name>:** é um nome único que é dado ao recurso;
- **description:** descrição do *output*;
- **value:** é o valor do recurso que normalmente é obtido através da função `get_attr`;
- **condition:** permite definir a condição com que o valor de *output* é apresentado;

3.1.4.2 Ceilometer (Telemetry)

Este é o serviço de telemetria que tem a tarefa de recolher os dados que os vários componentes do OpenStack fornecem. Inicialmente o Ceilometer foi desenvolvido apenas para recolher as informações necessárias para faturação da utilização dos recursos pelos utilizadores. Mas devido a uma necessidade de controlo do *cluster* percebeu-se que era necessário um serviço que fosse capaz de tirar medições gerais de todos os componentes do OpenStack. Estas medições, que também são chamadas de amostras, podem ser utilizadas para monitorização do sistema de maneira a despoletar ações ou alarmes. Por exemplo, lançar um aviso que o CPU de uma determinada instância está a ser utilizado a cima de um dado *threshold*, ou então quando os recursos de

um determinado grupo de instâncias está perto de se esgotar podem ser despoletadas ações de alocação de recursos para esse grupo.

3.2 Interfaces via linha de comandos com a *cloud*

Na Secção 3.1 foi abordado o sistema OpenStack, no qual, foi possível reunir conhecimento suficiente sobre os diferentes mecanismos e serviços que este fornece aos seus utilizadores.

Nesta dissertação como foi falado na Secção 1.2 é necessário explorar os vários componentes do OpenStack para criar grupos de instâncias com escalabilidade horizontal (conceito abordado na Secção 2.4) de maneira a trazer estas características elásticas ao *cluster* de gestão de contentores Kubernetes.

Estas funcionalidades de elasticidade serão integradas na CLI de gestão da *cloud* interna da AlticeLabs chamada pcloud. Devido a isso nesta secção vamos abordar as CLI de duas *clouds* públicas que são a Google Cloud Plataform e a Amazon AWS para servir de inspiração e exemplo para o desenvolvimento dos comandos de escalabilidade automática a integrar na pcloud.

De forma a perceber o estado de implementação da ferramenta pcloud no final desta secção será abordada a sua arquitetura, a sintaxe e a organização dos comandos que a compõe.

3.2.1 CLI: Google Cloud Platform

A Google é uma empresa bastante conhecida que não necessita de apresentações. Esta empresa tem inúmeros serviços disponíveis para os seus clientes. Um destes serviços é o Google Cloud Platform [57]. Este é composto por uma plataforma de *cloud* pública que fornece serviços ao nível do PaaS/IaaS tais como computação, armazenamento de dados, rede, *big data*, base de dados, *machine learning*, entre outros. Uma das formas de interagir e aprovisionar estes recursos é através da ferramenta gcloud.

O gcloud [58] é uma *Command Line Interface* (CLI) pertencente ao grupo de ferramentas Google Cloud SDK [59] que fornece uma interface com a Google Cloud Platform de maneira a realizar as diferentes tarefas nessa plataforma. Essas tarefas passam por lançar máquinas virtuais, instâncias de SQL, orquestração de *clusters*, gestão de *deployments*, entre outras.

Comandos Google Cloud Platform

O gcloud é composto por diferentes comandos organizados em grupos e por *flags* que permitem especificar as ações realizadas nos grupos de comandos. Os comandos e as *flags* disponíveis são organizados da seguinte forma [60]:

```
$ gcloud GROUP | COMMAND
    [--account=ACCOUNT]
    [--configuration=CONFIGURATION]
```

```
        [--flatten=[KEY,...]]
        [--format=FORMAT]
        [--help]
        [--project=PROJECT_ID]
        [--quiet, -q]
        [--verbosity=VERBOSITY; default=``warning''] [-h]
        [--version, -v]
        [--log-http]
        [--trace-token=TRACE_TOKEN]
        [--no-user-output-enabled]
```

Através do gcloud é possível manipular os seguintes grupos (`GROUP`): *app*, *auth*, *components*, *compute*, *config*, *container*, *dataflow*, *dataproc*, *datastore*, *debug*, *deployment-manager*, DNS, IAM, *organizations*, *projects*, *service-management*, *source*, SQL. Dos grupos enumerados anteriormente apenas nos vamos focar no *compute*, pois é aquele que tem mais interesse nesta dissertação por ser o grupo de comandos que é capaz de criar e manipular máquinas virtuais pertencentes ao Google Compute Engine [61]. O comando `gcloud compute` permite criar e manipular recursos relativos a máquinas virtuais, tais como discos, *firewall*, rede, escalabilidade, performance permitindo facilmente gerir um cluster completo. Mais uma vez e tendo em conta os objetivos desta dissertação, apenas vamos nos focar nos grupos de ações que permitem criar máquinas virtuais e manipular a escalabilidade horizontal de grupos de máquinas virtuais. De seguida serão mostrados os comandos e as acções necessárias para criar e manipular grupos elásticos de máquinas virtuais na Google Cloud Platform.

Escalabilidade Automática Google Cloud Platform

Para obter um grupo de máquinas elástico através do comando gcloud é necessário realizar duas ações:

- Criar um grupo de máquinas virtuais;
- Adicionar parâmetros de auto-escalabilidade ao grupo criado anteriormente;

Para criar um grupo de máquinas virtuais (instâncias) utilizamos o seguinte comando [62]:

```
$ gcloud compute instance-groups managed create example-group --zone
    us-central --template instance-template --size 1
```

O comando anterior cria um grupo chamado de `example-group` composto por `1` máquina virtual na zona `us-central`, em que cada uma das máquinas virtuais utilizará o *template* `instance-template` para a sua configuração inicial.

Visto que o grupo de máquinas virtuais criado anteriormente tem um número fixo de 1 elemento, mas como queremos que este valor seja dinâmico e escalável é necessário realizar um comando extra que irá definir parâmetros de auto-escalabilidade no grupo. Esses parâmetros são definidos à custa da seguinte sintaxe de comandos [63]:

```
$ gcloud compute instance-groups managed set-autoscaling NAME
      --max-num-replicas=MAX_NUM_REPLICAS
      [--cool-down-period=COOL_DOWN_PERIOD]
      [--custom-metric-utilization=[metric=METRIC]
      [utilization-target=UTILIZATION-TARGET]
      [utilization-target-type=UTILIZATION-TARGET-TYPE]]
      [--description=DESCRIPTION]
      [--min-num-replicas=MIN_NUM_REPLICAS]
      [--scale-based-on-cpu]
      [--scale-based-on-load-balancing]
      [--target-cpu-utilization=TARGET_CPU_UTILIZATION]
      [--target-load-balancing-utilization=TARGET_LOAD_BALANCING_UTILIZATION]
      [--region=REGION --zone=ZONE]
```

Este comando é composto pelo argumento `NAME` que é o nome do grupo a que queremos adicionar os parâmetros de auto-escalabilidade e por um conjunto de *flags*. A *flag* `–max-num-replicas` é obrigatória e define o número máximo de réplicas que o grupo terá na eventualidade de ações de escalabilidade. Para além desta *flag* existe um grupo de *flags* opcionais, que são:

- `--cool-down-period`: O período de tempo até iniciar a coleta de informações (métricas) de uma nova instância. Previne a extração de informação quando a instância está a iniciar. Valor *default* 60s.
- `--custom-metric-utilization`: Define qual a métrica das instâncias a avaliar.
- `--description`: Permite adicionar notas ao *autoscaler*.
- `--min-num-replicas`: Número mínimo de instâncias que o grupo terá.
- `--scale-based-on-cpu`: As atividades de escalabilidade serão despoletadas com base na percentagem da utilização de CPU.
- `--scale-based-on-load-balancing`: As atividades de escalabilidade serão despoletadas com base na percentagem da utilização do balanceador de carga.
- `--target-cpu-utilization`: Valor de utilização de CPU ao qual o *autoscaler* terá como objetivo manter.
- `--target-load-balancing-utilization`: Valor de utilização do balanceador de carga ao qual o *autoscaler* terá como objetivo manter.
- `--region, --zone`: Apenas uma destas *flags* poderá ser especificada. `--region` permite especificar a região na qual o grupo será operado. `--zone` permite especificar a zona na qual o grupo será operado.

3.2.2 CLI: Amazon AWS

À semelhança da Google Cloud Platform apresentada anteriormente a empresa Amazon também dispõe de um serviço de IaaS chamado Amazon Elastic Compute Cloud (ou Amazon EC2) [64]. Este é um serviço de *cloud* pública que disponibiliza recursos computacionais elásticos direcionados essencialmente para entidades que vendem produtos a partir da *web*. Uma das

formas de manipular os recursos disponíveis na *cloud* da Amazon é a partir da CLI da *Amazon Web Services* (AWS) [65]. Esta CLI tem um conjunto de comandos que manipulam todos os recursos da *cloud*. Mas os comandos que têm maior valor para esta dissertação são aqueles que permitem manipular grupos elásticos de instâncias e como tal serão falados de seguida.

Escalabilidade Automática AWS EC2

A função associada à linha de comandos AWS que fornece elasticidade e escalabilidade automática é o `autoscaling` [66]. Este grupo de comandos tem a funcionalidade de lançar ou terminar instâncias com base em regras, horários ou diagnósticos (*health checks*). É composto por um grande número de subcomandos que desempenham ações ao nível do acoplamento de instâncias, gestão de balanceadores de carga, configurações, políticas, notificações, métricas, *tags* e criação de grupos de auto-escalabilidade.

A sintaxe associado ao comando `create-auto-scaling-group` que tem a tarefa de criar grupos de auto-escalabilidade é a seguinte [67]:

```
$ aws autoscaling create-auto-scaling-group
       --auto-scaling-group-name <value>
       [--launch-configuration-name <value>]
       [--instance-id <value>]
       --min-size <value>
       --max-size <value>
       [--desired-capacity <value>]
       [--default-cooldown <value>]
       [--availability-zones <value>]
       [--load-balancer-names <value>]
       [--target-group-arns <value>]
       [--health-check-type <value>]
       [--health-check-grace-period <value>]
       [--placement-group <value>]
       [--vpc-zone-identifier <value>]
       [--termination-policies <value>]
       [--new-instances-protected-from-scale-in]
       [--tags <value>]
       [--cli-input-json <value>]
       [--generate-cli-skeleton <value>]
```

Como foi visto anteriormente este comando reúne um conjunto de *flags*. De seguida vamos detalhar a especificação que cada uma das *flags* do comando:

- `--auto-scaling-group-name`: Define o nome do grupo.
- `--launch-configuration-name`: Define o nome da configuração de inicialização do grupo.
- `--instance-id`: Define o identificador da instância que irá lançar a configuração do grupo.
- `--min-size`: Define o tamanho mínimo do grupo.

- `--max-size`: Define o tamanho máximo do grupo.
- `--desired-capacity`: Define o número de instâncias que devem estar executar no grupo.
- `--default-cooldown`: Define a quantidade de tempo, em segundos, entre atividades de auto-escalabilidade. O valor *default* é 300 s.
- `--availability-zones`: Define as zonas onde o grupo deve operar.
- `--load-balancer-names`: Define um ou mais balanceadores a utilizar para o balanceamento de carga pelo grupo.
- `--target-group-arns`: Define a lista dos *Amazon Resource Names*.
- `--health-check-type`: Define o serviço externo a utilizar para realizar os *health checks* às instâncias.
- `--health-check-grace-period`: Define a quantidade de tempo em segundos até realizar o primeiro *health check*.
- `--placement-group`: Define o nome do grupo lógico onde as instâncias serão lançadas.
- `--vpc-zone-identifier`: Define a *subnet* ao qual as instâncias se vão ligar.
- `--termination-policies`: Define uma ou mais regras para terminar as instâncias.
- `--new-instances-protected-from-scale-in`: Indica as instâncias que estão protegidas de serem eliminadas devido a atividades de auto-escalabilidade.
- `--tags`: Permite definir uma ou mais tags.
- `--cli-input-json`: Permite a realização de tarefas/ações definidas no formato JSON.
- `--generate-cli-skeleton`: Valida e imprime para o *standard output* o esqueleto JSON sem enviar um pedido API.

3.2.3 CLI: pcloud

O pcloud é uma interface cliente via linha de comandos que está ser desenvolvida na AlticeLabs [4], na linguagem de programação Go [68]. Este cliente tem como objetivo simplificar o controlo e gestão da plataforma OpenStack nas equipas de desenvolvimento e produção de *software* da AlticeLabs. Como foi visto na Secção 3.1, um utilizador para interagir com a plataforma OpenStack tem três opções: via DashBoard Horizon, via linha de comandos CLI e via API. Para manipular e utilizar cada uma das opções anteriores é necessário um conhecimento alargado da infraestrutura da *cloud*, dos projetos/serviços que estão por trás do Openstack (como por exemplo Nova, Neutron, etc.) e conceitos intrinsecamente ligados apenas ao OpenStack (como por exemplo, o conceito de Floating IP). Para fornecer este conhecimento às equipas de desenvolvimento e produção de *software* era necessária uma grande quantidade de tempo e além do mais estaríamos a desviar as equipas do *core* das tarefas que têm a desempenhar.

Esta *framework* nasce com o objetivo de reunir um conjunto de funcionalidades simples que criam e gerem as diferentes infraestruturas necessárias durante todo o ciclo de vida dos produtos de *software* na *cloud* privada da AlticeLabs.

Até à data, o pcloud reúne várias opções para manipular os recursos da cloud. Estas opções

são:

- Criar, eliminar, listar, parar, iniciar servidores;
- Tirar *snapshots* e reverter servidores para imagens de *snapshots*;
- Informação dos recursos disponíveis para o projeto;
- Informação das imagens e *flavors* disponíveis para o projeto;
- Gestão de utilizadores e acessos;

Nesta secção vamos apresentar de forma introdutória a *framework* pcloud, descrevendo a sua arquitetura, as suas funcionalidades e a sintaxe dos comandos que constituem o pcloud.

Arquitetura

Como foi apresentado no diagrama 3.1 o OpenStack dispõe de uma API pela qual é possível invocar todas as funcionalidades disponíveis. É neste estilo que todos os outros serviços do OpenStack desempenham as suas funções. Por exemplo o DashBoard Horizon interage com a API do Openstack de maneira a desempenhar todas as suas funcionalidades. A mesma estratégia é utilizada na *framework* pcloud.

Comunicação (Gophercloud)

De maneira a facilitar a comunicação com a API do Openstack é utilizado o *Software Development Kits* (SDK) Gophercloud [69]. Este SDK totalmente *open source* é desenvolvido pela Rackspace [70] na linguagem de programação Go [68]. O Gophercloud serve de intermediário das comunicações entre o pcloud e a API do OpenStack. O que faz com que a comunicação entre as duas plataformas seja simples do ponto de vista do programador. O GopherCloud tem funcionalidades com as API's dos seguintes projetos do OpenStack: Compute, Neutron, Cinder, Keystone, Ceilometer e Heat.

Apresentação CLI (Cobra)

Para criar a linha de comandos e gerir a apresentação dos dados é utilizado a biblioteca Cobra [71]. Esta biblioteca tem o objetivo de auxiliar a criação de interfaces de linhas de comando modernas (CLI), tais como as CLI do Kubernetes, do Docker, do etcd, e entre outros. As principais funcionalidades desta ferramenta são:

- Gestão de comandos e subcomandos;
- Suporte para flags locais e globais;
- Integração automática de sugestões inteligentes;
- Criação automática das *help pages*;
- Criação automática do *autocomplete* dos comandos;
- *Aliases* de comandos;

As CLI criadas com o Cobra seguem uma estrutura organizada em: comandos, argumentos e *flags*. No contexto de desenvolvimento os comandos representam ações, os argumentos são

objetos e as *flags* são modificadores das ações.

No seguinte exemplo o `server` é um comando, o `create` é um argumento e o `--port` é uma *flag*:

```
> hugo server create --port=1313
```

Os comandos são o ponto central da aplicação, pois cada interação suportada será um comando do Cobra. Estes são organizados numa estrutura em árvore. As *flags* são uma forma de modificar ou especificar o comportamento de um comando. Estas *flags* podem ser locais apenas a um comando ou então globais a todos os comandos.

Ficheiros de Configuração

O pcloud de maneira a definir as suas configurações *default* mantém um ficheiro denominado `pcloud.conf` em que o seu conteúdo é o seguinte:

```
"PCLOUD_URL": "Identity_Service",
"PCLOUD_NET": "Default_Network",
"PCLOUD_FLAVOR": "Default_Flavour",
"PCLOUD_IMAGE": "Default_Image",
```

No campo `PCLOUD_URL` é definido o *endpoint* do serviço Identity do Openstack. Este *endpoint* serve para que o pcloud consiga obter uma *token* de maneira a autenticar-se com o Openstack. O campo `PCLOUD_NET` permite definir a rede *default* ao qual os `servers` se vão ligar. O campo `PCLOUD_FLAVOR` define o *flavour* (conceito abordado na Secção 3.1.3.1) *default* que será atribuído aos `servers` e o campo `PCLOUD_IMAGE` define a imagem *default* que os `servers` criados a partir do pcloud vão utilizar.

Para além do ficheiro `pcloud.conf` existe outro ficheiro denominado `token` que permite especificar uma *token* alternativa além da *token* atribuída no momento de *login* inicial.

Sintaxe

Como foi abordado anteriormente, o pcloud é uma interface via linha de comandos que corre em ambientes Linux. Estes comandos seguem uma estrutura que é divida em objetos, ações sobre esses objetos e *flags* para especificação e modificação de ações:

```
$ pcloud [objeto] <acao> [--flags]
```

De seguida será apresentada a árvore completa da organização e disposição dos argumentos, comandos e flags do pcloud:

```
pcloud
├── login
├── logout
├── server
│   ├── start
│   ├── stop
│   ├── restart
│   │   └── --force
│   ├── snapshot
│   │   ├── --force
│   │   ├── --public
│   │   └── --snapname
│   ├── revert
│   │   ├── --force
│   │   └── --snapname
│   ├── list
│   │   └── -l, --long
│   ├── delete
│   │   └── --force
│   └── launch
│       ├── --flavor
│       └── --image
├── image
│   ├── list
│   │   ├── -l, --long
│   │   └── -public
│   ├── delete
│   └── set
│       └── -v,--visibility
├── user
│   ├── grant
│   │   └── --role
│   ├── revoke
│   │   └── --role
│   └── list
├── flavor
│   └── list
├── usage
├── -c, --config
└── -t, --token
```

Como podemos verificar pela árvore de comandos anteriormente representada os objetos que conseguimos manipular através dos comandos do pcloud são:

login

```
$ pcloud login <projeto> <username>
```

O comando "pcloud login" serve para obter uma *token* de sessão para um determinado `<projeto>` dado um `<username>` específico.

logout

```
$ pcloud logout [flags]
```

O comando "pcloud logout" permite terminar a sessão corrente, eliminando a *token* anteriormente recebida.

server

```
$ pcloud server <acao>
```

As ações que podemos desempenhar numa máquina virtual (`server`) são:

- `start <hostname>`: Inicia um `server`.
- `stop <hostname>`: Pára um `server`.
- `restart <hostname> [flags]`: Reinicia um `server`.
  - `--force`: Força o reiniciar mesmo que o `server` esteja em execução.
- `snapshot <hostname> [flags]`: Cria um *snapshot* de um `server`.
  - `--force`: Tira o *snapshot* mesmo que o `server` esteja em execução.
  - `--public`: O *snapshot* criado será público.
  - `--snapname`: Nome do *snapshot*.
- `revert <hostname> [flags]`: Reverte um `server` a partir de um *snapshot*.
  - `--force`: Força o reverter mesmo que o `server` esteja em execução.
  - `--snapname`: Nome do *snapshot* a utilizar.
- `list [servername] [flags]`: Lista os `server` existentes.
  - `-l, --long`: Mostra informação adicional.
- `delete <servername> [flags]`: Elimina um `server`.
  - `--force`: Força o eliminar mesmo que o `server` esteja em execução.
- `launch <servername> [flags]`: Lança um `server`.
  - `--flavor`: Especifica o `flavor` a utilizar no `server`.
  - `--image`: Especifica a imagem do SO a utilizar no `server`.

image

```
$ pcloud image <acao>
```

As ações que podemos desempenhar numa imagem (`image`) são:

- `list [imagename] [flags]`: Lista as `images` disponíveis.
  - `-l, --long`: Mostra informação adicional.
  - `--public`: Lista as `image` públicas.
- `delete <imagename> [flags]`: Elimina uma `image` (snapshot).
- `set <imagename> [flags]`: Atualiza as propriedades de uma `image`.
  - `-v, --visibility`: Atualiza a visibilidade (pública ou privada) da `image`.

user

```
$ pcloud user <acao>
```

As ações que podemos desempenhar num utilizador (`user`) são:

- `grant <username> [flags]`: Concede uma função (exemplo: `member`) a um `user` no projeto corrente.
  - `--role`: Específica a função (exemplo: *admin* ou *member*).
- `revoke <username> [flags]`: Revoga uma função a um `user` no projeto corrente.
  - `--role`: Específica a função (exemplo: *admin* ou *member*).
- `list [username]`: Lista os `users` do projeto corrente.

flavor

```
$ pcloud flavor <acao>
```

A ação que podemos desempenhar num `flavor` é:

- `list`: Lista os *flavors* existentes.

usage

```
$ pcloud usage
```

O comando "pcloud usage" permite obter informação à cerca da utilização dos recursos alocados ao projeto.

3.3 Plataforma de Conteinerização - Docker

Docker [2] é uma plataforma *open-source* de conteinerização de aplicações que surge como uma alternativa leve à utilização de máquinas virtuais. Para além disso este simplifica o processo de instalação, execução, publicação e remoção de *software*. O Docker não criou o conceito de contentores (falado anteriormente na secção 2.2.3) mas torna mais acessível a utilização dos mesmos.

Ao contrario das máquinas virtuais o Docker permite que processos possam ser executados de uma forma isolada partilhando o *kernel* da *host*. Este isolamento faz com que seja possível executar qualquer versão de mesmo *software* na mesma máquina *host*. Outro aspeto que este isolamento traz é a organização das aplicações e das suas dependências. Devido ao isolamento todas as aplicações e as suas dependências são organizadas e correm dentro do seu contentor. Como as dependências correm dentro dos contentores aumenta a portabilidade das aplicações permitindo que uma aplicação possa correr em qualquer versão do sistema operativo Linux. Esta portabilidade também é prolongada aos sistemas operativos Windows e OS X, mas para isso é necessário utilizar apenas uma única máquina virtual.

Nesta secção vamos abordar a plataforma de conteinerização de aplicações Docker, expondo os seus conceitos fundamentais assim como a sua arquitetura.

3.3.1 Arquitetura

A plataforma Docker, também conhecida como Docker *engine*, é uma aplicação do tipo cliente-servidor [72]. O servidor é chamado de Docker *daemon* que é o processo pai de todos os processos a correr em contentores, desta forma dizemos que o Docker *daemon* é responsável pela gestão dos contentores. Enquanto que o cliente é uma CLI que interage com o *daemon* através de pedidos à API.

3.3.2 Imagens Docker

Uma imagem Docker pode ser considerada uma forma de empacotamento de *software* e suas dependências de maneira a que possa ser movida entre vários ambientes e executada sem alterações. Tendo em conta este conceito podemos dizer que um contentor é a instanciação de uma imagem.

As imagens Docker podem ser criadas de duas formas, através de um *snapshot* de um contentor em execução ou através de um ficheiro de *build* chamado Dockerfile.

Dockerfile

Dockerfile é um ficheiro que tem um conjunto de instruções ao qual o Docker é capaz de ler e com base nessas instruções criar automaticamente uma imagem. Uma imagem Docker é constituída por um conjunto de camadas. Cada uma destas camadas representa uma instrução na Dockerfile da imagem. O conceito de camadas será falado na próxima secção.

No exemplo a seguir é mostrado a criação de uma imagem com o sistema de controlo de versões Git através de uma Dockerfile.

```
CAMADA 0:    FROM ubuntu:latest
CAMADA 1:    LABEL maintainer="dreis@fcup.com"
```

```
CAMADA 2:    RUN apt-get install -y git
CAMADA 3:    ENTRYPOINT ["git"]
```

A partir deste ficheiro basta executar o comando "`docker build -tag git:latest .`" que é iniciado um processo automático de construção da imagem.

A Dockerfile exemplificada é constituída por quatro instruções, ao qual cada um delas cria uma camda na imagem, que são:

- `FROM`: A instrução `FROM` informa o Docker para começar o processo de *build* a partir de uma imagem do Ubuntu na sua versão *latest*.
- `LABEL`: Esta instrução permite adicionar meta informação à imagem.
- `RUN`: A instrução `RUN` permite executar qualquer tipo de comando. No caso é executado o comando apt-get para instalar o Git.
- `ENTRYPOINT`: Indica o comando de arranque do contentor, ou seja o comando de inicialização.

Uma lista com todas as instruções que podem ser utilizadas numa Dockerfile pode ser consultada em [73].

3.3.3 Camadas

Como foi dito anteriormente uma imagem Docker é constituída por um conjunto de camadas. Este conjunto de camadas que constituem uma imagem formam uma pilha de camadas, ao qual cada uma destas camadas é um sistema de ficheiros. Na base desta pilha de camadas encontramos o sistema de ficheiros de *boot* (*bootfs*) e logo de seguida encontramos o sistema de ficheiros de *root* (*rootfs*) [74]. Para melhor exemplificar a pilha de camadas que formam uma imagem Docker é apresentado o diagrama da Figura 3.2.

Um contentor Docker quando é lançado é constituído pelo conjunto de camadas que formam a imagem (com permissões apenas de leitura) e com uma camada adicional em cima de todas as outras (com permissões de leitura e escrita), como pode ser visto pela Figura 3.2. Esta última camada quando é lançada no momento de arranque do contentor encontra-se vazia. Sempre que são realizadas alterações aos ficheiros das camadas mais abaixo esses ficheiros são copiados para a camada superior de maneira a registar essas alterações. Este processo de manter várias versões do mesmo ficheiro distribuído pelas várias camadas é chamado de "*Copy-on-write*" falado na próxima secção.

A interação e a união dos vários sistemas de ficheiros que formam a pilha de camadas é realizada por um *storage driver* [75]. Este *storage driver* depende do sistema operativo da *host*, por exemplo no caso do Ubuntu é utilizado o sistema de ficheiros de união AUFS e no caso do CentOS é utilizado a *framework devicemapper*.

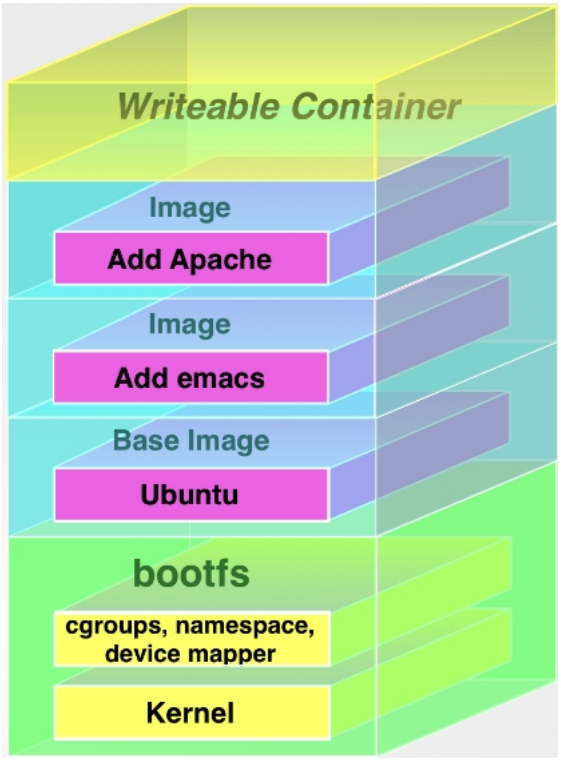


Figura 3.2: Exemplo de uma imagem Docker e as suas camadas. (Imagem extraída de [74])

Estratégia "*copy-on-write*" e Reutilização de Camadas

Copy-on-write [75] é uma estratégia de partilha e cópia de ficheiros que aumenta a eficiência no armazenamento e transferência de imagens através da reutilização de camadas entre várias imagens. Esta estratégia permite que quando a camada superior que está em execução num contentor necessitar de ler um ficheiro que está na camada inferior lê diretamente esse ficheiro da camada inferior. Mas na primeira vez que a camada superior necessitar de escrever num ficheiro da camada inferior esse ficheiro é copiado para camada superior onde será realizada a alteração. Isto permite a partilha de camadas entre imagens, o que faz com que quando queremos transferir uma imagem em que parte das camadas que compõe essa imagem já existem localmente no sistema apenas serão transferidas as camadas em falta.

3.3.4 Repositório de Imagens: Docker Registry

As imagens depois de serem criadas à custa das Dockerfiles têm de ser armazenadas num repositório para que possam ser distribuídas, para isso é utilizado o Docker *Registry*. O Docker disponibiliza um *Registry* público que é chamado de Docker Hub [76]. Como este repositório é público permite que as imagens desenvolvidas pelos programadores possam ser partilhadas por todos.

Para além disso é possível criar *Registrys* privados no Docker Hub com custos associados. O

Docker *Registry* é *open-source* o que permite que as organizações possam criar repositório de imagens privados no seu *data center* [77].

3.4 Gestão de Contentores em Cluster

Como vimos anteriormente, na Secção 2.2, as tecnologias de virtualização baseadas em contentores trazem várias vantagens, tanto para o provedor de serviços da *cloud*, como para os produtores de *software*. Pois são uma opção leve quando comparados com as máquinas virtuais e que fazem um uso mais eficiente da infraestrutura subjacente permitindo o *deployment* de múltiplos contentores num único nó. Além disto, tornam a execução de aplicações na *cloud* uma tarefa mais flexível, pois através dos contentores é mais simples e rápido lançar aplicações, assim como, lidar com uma demanda flutuante de pedidos e requisitos por parte dos clientes. Outro ponto positivo na utilização de contentores é ao nível da portabilidade das aplicações. Ou seja, estes tornam mais simples movimentar aplicações entre diferentes ambientes.

Com a criação de aplicações em arquiteturas orientadas ao serviço e micro serviços implementas à custa de dezenas ou até centenas de contentores distribuídos por várias máquinas torna a sua gestão e *deployment* numa tarefa complexa. É neste ponto que necessitamos de ferramentas de orquestração de contentores, de maneira a gerir e controlar esses contentores a partir de um ponto central. Este tipo de ferramentas têm a tarefa de controlar e gerir grandes quantidade de contentores distribuídos por um *cluster* sem grandes intervenções manuais. Para isso é necessário que estes orquestradores tenham mecanismos que sejam capazes de lidar com falhas tendo em conta certas políticas.

Nesta secção vamos abordar as ferramentas de orquestração e gestão de contentores em *cluster* mais conhecidas, que são o caso do Mesos, Docker Swarm e Kubernetes. Com isto espera-se que seja possível perceber as razões que levaram a selecionar o Kubernetes como a plataforma para a gestão de aplicações conteinerizadas nesta dissertação (este assunto será bordado na Secção 4.6). Para cada uma destas ferramentas vamos abordar a sua arquitetura, funcionalidades e conceitos. Como a ferramenta Kubernetes faz parte dos objetivos desta dissertação esta será abordada com uma maior profundidade.

3.4.1 Apache Mesos

O Apache Mesos [78] é um *software open-source* inicialmente desenvolvido na Universidade da Califórnia, Berkeley, no ano de 2009. A sua principal função é a gestão de *clusters* num formato *standard* que abstrai um *cluster* de máquinas para uma única máquina gigante, na qual é capaz de correr *frameworks* (como por exemplo, Hadoop, Spark, Kafka, Marathon) nesse ambiente distribuído.

Este segue uma arquitetura *master/slave* como se pode ver pela Figura 3.3. O processo *master* distribui os recursos do *cluster* pelas *frameworks* e os processos *slave* instalados em cada

nó do *cluster* realizam as tarefas das *frameworks*.

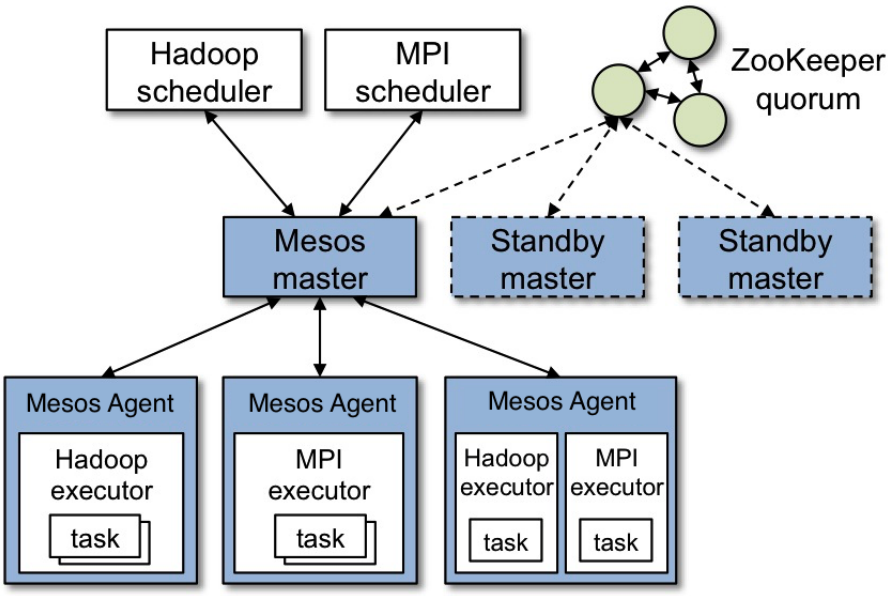


Figura 3.3: Arquitetura do Mesos. (Imagem extraída de [79])

A *framework* que permite o *deployment* e gestão de aplicações conteinerizadas no Mesos é chamada de Marathon [80]. Esta *framework* faz o *deployment* através de ficheiros JSON de contentores Mesos ou Docker nos nós *slave* e fornece uma API para iniciar, parar e escalar os serviços. Esta *framework* tem as seguintes funcionalidades [81]:

- **Alta Disponibilidade:** É possível que uma das instâncias da Marathon possa falhar que o serviço continua a estar disponível. Esta funcionalidade é concretizada com a instalação e configuração do *software* Zookeeper;
- **Aplicações com estado:** permite albergar aplicações com estado (por exemplo bases de dados) fornecendo armazenamento persistente à aplicação. Esta funcionalidade ainda está numa fase *Beta* e a sua utilização num ambiente de produção pode levar a falhas da aplicação [82];
- **Interface gráfica:** gestão de contentores e do *cluster* a partir de uma interface gráfica;
- **Restrições:** permite restringir contentores a determinados recursos ou nós do *cluster*;
- **Descoberta de Serviços e Balanceamento de Carga:** a descoberta de serviços é realizado através do Mesos DNS e o balanceamento de carga é realizado através da instalação do balanceador *open-source* HAProxy [83]. Esta solução provoca um *delay* significativo na disponibilidade do serviço [84];
- **Controlo da aplicação:** controlo da aplicação através de *health checks*. Para isso a aplicação terá de ter um porto à escuta de maneira a realizar os pedidos de *health check*;
- **Métricas:** obtenção de métricas dos recursos através de http;
- **API REST:** com esta API é possível aumentar a integração do Marathon com outras plataformas ou a utilização de *scripts* para a realização de certas tarefas cíclicas.

3.4.2 Docker Swarm

Docker Swarm [85] é a ferramenta nativa do Docker para a orquestração de contentores em *cluster*. Este novo complemento ao Docker *engine* vem fornecer capacidades nativas de gestão de contentores em *cluster* de forma simples. Isto permite que um grupo de Docker *engines* possam ser geridos como se de apenas um elemento se tratasse, ou seja, permite escalar aplicações como se estivessem a ser executadas numa única máquina. À semelhança dos outros orquestradores de contentores estudados nesta dissertação, o Docker Swarm segue uma arquitetura dividida em dois componentes fundamentais os nós *master* que são os Swarm *managers* e os nós *slave* que são os nós *worker* responsáveis pela execução dos contentores através do Docker *engine*, como se pode ver pela Figura 3.4.

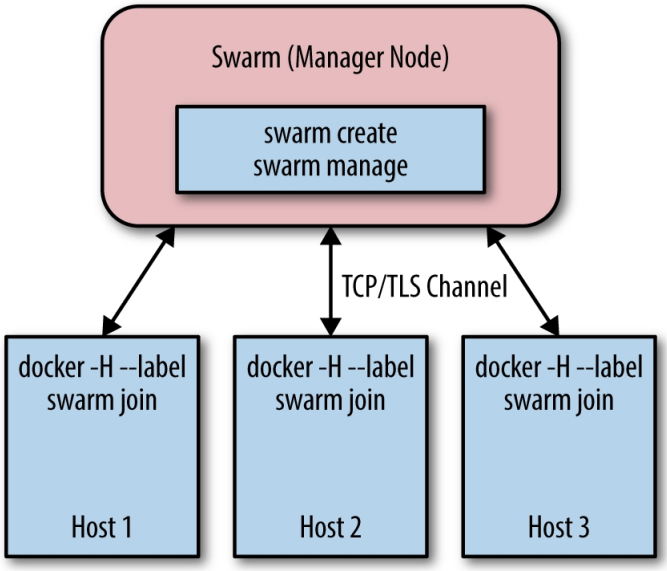


Figura 3.4: Arquitetura do Docker Swarm. (Imagem extraída de [86])

O Docker Swarm tem as seguintes funcionalidades implementadas [87]:

- **Escalabilidade:** permite que o número de contentores dedicados a uma tarefa possam ser aumentados ou diminuidos;
- **Adaptação do estado desejado:** o Swarm *manager* realiza uma monitorização constante do *cluster* e sempre que deteta alterações ao estado pré-definido desejado realiza as devidas alterações de maneira a colocar o *cluster* nesse estado desejado;
- **Rede multi-host:** é possível criar uma rede *overlay* para os serviços da aplicação fornecendo endereços IP a cada contentor;
- **Descoberta de serviços:** por cada serviço existente é criado um nome único no DNS, ao qual é possível balancear o tráfego com base nesse nome;
- **Balanceamento de carga:** é possível expor portas dos serviços internos a um balanceador externo. O balanceamento interno é responsabilidade do Docker Swarm;
- **Seguro:** cada nó do *cluster* comunica de forma encriptada pelo protocolo de segurança TLS;

- **Gestão de atualizações**: permite a realização de atualizações com base em políticas.

O *deployment* de aplicações compostas por vários contentores é feito à custa de uma ferramenta extra chamada Docker Compose [88]. Esta ferramenta utiliza ficheiros de configuração YAML que permitem configurar e interligar os contentores da aplicação criando os vários serviços associados à aplicação.

3.4.3 Kubernetes

Kubernetes ou K8s [3] é uma plataforma *open-source* lançado em 2014 pela Google que representa a sua experiência de mais de uma década na gestão de aplicações conteinerizadas. Após o seu lançamento, várias organizações interessadas como Red Hat, CoreOS, CERN, etc, realizaram algumas contribuições que aumentaram e melhoraram as capacidades desta plataforma tornando-a mais robusta e preparada para o ambiente de produção [89]. Esta ferramenta neste momento está sobre a alçada da Cloud Native Computing Fondation que sugere a sua utilização [90] como a ferramenta *standard de facto* da orquestração de contentores.

O Kubernetes é capaz de automatizar e gerir vários elementos de uma aplicação conteinerizada durante o seu ciclo de vida, como por exemplo, o *deployment*, descoberta de serviços, gestão de configurações, balanceamento de carga, elasticidade, atualizações, armazenamento persistente, monitorização e *logging* [91]. Para além disso, o Kubernetes é capaz de monitorizar todo o *cluster* e no caso de falha de algum contentor este é capaz de relançar esse contentor noutro nó do *cluster*. Ou até mesmo no caso de um dos nós do *cluster* falhar o Kubernetes é capaz de relançar todos os contentores que estavam nesse nó noutro nó saudável do *cluster*.

3.4.3.1 Objetos

Todos elementos do Kubernetes são tratados por objetos e é destes objetos que vamos falar nesta secção.

Pod

O Pod é uma abstração que representa um grupo de um ou mais contentores Docker que correm em conjunto, com ou sem volumes. Nesta abordagem deve existir um contentor principal com apenas um processo e todos os restantes contentores que estão no Pod apenas devem auxiliar a execução do contentor principal [92]. Os vários contentores que correm em conjunto no mesmo Pod estão num ambiente de partilha de recursos e podem comunicar via *localhost*.

Estes têm um ciclo de vida associado que podem passar pelos seguintes estados: pendente, em execução, sucedido, falha, desconhecido [93]. Assim que um Pod é lançado num nó do *cluster* este permanece lá até que algo o destrua. Se um Pod entrar num estado de falha este é eliminado e um novo Pod é relançado, possivelmente noutro nó do *cluster*.

Volume

No Kubernetes um Volume é um diretório da *host* que é montado no Pod. O ciclo de vida de um Volume é o mesmo que o ciclo de vida de um Pod. Isto faz com que quando um contentor é reiniciado após uma falha os dados são preservados, mas caso um Pod seja eliminado o Volume também será eliminado [94]. Para além disso o Volume permite a partilha de ficheiros entre contentores que pertencem ao mesmo Pod.

Service

Services é outra camada de abstração que define um conjunto de Pods semelhantes (réplicas) que fornecem um serviço. Como os Pods são uma unidade efémera, que podem ter um ciclo de vida curto, os pedidos de comunicação não podem ser enviados diretamente do remetente para o Pod. Para isso é utilizado os Services como um competente consistente que lida com a distribuição de pedidos pelos vários Pods [95]. Então através dos Services é possível expor os Pods a tráfego interno e externo ao *cluster* com balanceamento de tráfego pelas várias réplicas. A cada Service é associado um endereço IP e um nome que permite que os outros Pods possam descobrir o serviço dentro do *cluster* Kubernetes.

Label

Label é um mecanismo de seleção e agrupamento constituído por pares chave/valor que são anexados a objetos (por exemplo Pods) e que servem para identificar e organizar objetos em grupos relacionados de uma forma significativa para o utilizador [96]. É a partir destas Labels que são selecionadas as réplicas de Pods que fazem parte de um Service.

ReplicaSet

O ReplicaSet é um objeto que evolui do Replication Controller e permite que um número especifico de réplicas de Pods estão a ser executadas em qualquer momento do tempo no *cluster* [97]. Ou seja, os Pods mantidos pelo ReplicaSet são automaticamente repostos caso falhem, sejam eliminados, ou terminados.

Deployment

Os parâmetros que definem os objetos Pod ou ReplicaSet são estáticos e permanecem durante todo o seu ciclo de vida, o que faz com que alterações ou atualizações a esses objetos sejam impossíveis. Os Deployments vem permitir lidar com esses problemas pois são um conceito de um nível mais alto em relação a esses objetos, que permitem gerir grupos de Pods ou ReplicaSets tornando os seus parâmetros dinâmicos e fornecendo atualizações declarativas.

Os Pods criados à custa do objeto Deployment olham para o estado desejado e definido pelo o utilizador e sempre que esse estado seja alterado o Kubernetes irá proceder às alterações necessárias até atingir esse novo estado desejado [98].

StatefulSet

StatefulSet é um objeto que permite controlar a ordem de *deployment* de um grupo de Pods e atribui identificadores estáveis a esses Pods. Neste objeto o *deployment* dos Pods é controlado, ou

seja, só passa para o *deployment* do Pod seguinte quando o *deployment* do Pod anterior tiver sido realizado com sucesso. Com a atribuição de identificadores estáveis permite que quando um Pod é relançado noutro nó receba invariavelmente o mesmo identificador [99]. Esta funcionalidade tem vantagens para o *deployment* de aplicações com estado criadas em *cluster* em que cada um dos elementos do *cluster* tem de conhecer os seus pares de forma a interagir diretamente com estes.

A atribuição dos identificadores de rede aos Pods é feita através de um mecanismo do Kubernetes chamado *Headless Service* [100]. Este tipo de objeto Kubernetes permite desacoplar os Pods do objeto Service para que os pedidos sejam enviados diretamente do remetente para o Pod, independentemente do nó do *cluster* onde se encontre pois o Pod terá sempre o mesmo identificador de rede durante o seu ciclo de vida.

3.4.3.2 Aplicações num Ambiente Kubernetes

O Kubernetes além de permitir o *deployment* e gestão de contentores tem um conjunto de funcionalidades que auxiliam a execução de aplicações em ambientes conteinerizados, tornando a tarefa de desenvolver aplicações com uma arquitetura baseada em micro-serviços ainda mais simples. Nesta secção vamos abordar esses mecanismos adicionais do Kubernetes.

Descoberta de Serviços

As aplicações em arquiteturas de micro-serviços compostas por vários contentores têm de ter uma forma de conseguir encontrar os outros serviços dentro do *cluster* de maneira a interagir. Como o ambiente subjacente que suporta aplicação é um *cluster* composto por várias máquinas, o que faz com que um contentor possa nascer em qualquer nó e devido a isso a aplicação não saber à priori o endereço que este irá receber, torna toda esta tarefa de descoberta de serviços ainda mais complexa.

Como foi visto anteriormente, um dos objetos que é possível criar no Kubernetes é os *Services*. Estes *Services* balanceiam tráfego por conjuntos de *Pods* que formam o serviço. Para que estes *Services* possam ser encontrados pelos outros *Pods* podem ser utilizadas duas estratégias, uma através de variáveis de ambiente e outra através do serviço de resolução de nomes DNS.

Na primeira estratégia, por cada serviço criado são introduzidas no arranque do contentor um conjunto de variáveis de ambiente com os respetivos endereços, como se pode ver no exemplo a seguir para o serviço chamado `EXEMPLO`:

```
EXEMPLO_SERVICE_HOST=10.0.0.11
EXEMPLO_SERVICE_PORT=6379
EXEMPLO_PORT=tcp://10.0.0.11:6379
EXEMPLO_PORT_6379_TCP=tcp://10.0.0.11:6379
EXEMPLO_PORT_6379_TCP_PROTO=tcp
EXEMPLO_PORT_6379_TCP_PORT=6379
EXEMPLO_PORT_6379_TCP_ADDR=10.0.0.11
```

A outra estratégia é a utilização do serviço DNS que resolve o nome atribuído ao Service para o seu endereço IP dentro do *cluster*. Por exemplo para o serviço chamado `exemplo` utilizávamos o endereço `exemplo.my-namespace`, em que o `my-namespace` é nome do *namespace* em que o serviço está atribuído. Para esta opção funcionar é necessário ter o *add-on* SkyDNS instalado no *cluster* Kubernetes [101].

Configurações

Alguns serviços conteinerizados necessitam de configurações para proceder ao seu arranque e subsequentes execuções. Para além disso as configurações podem variar nos seus parâmetros de ambiente para ambiente e para que as imagens sejam portáteis e não especificas a um ambiente estes ficheiros de configuração não constam do conteúdo da imagem. Para lidar com este problema o Kubernetes utiliza uma estratégia própria denominada ConfigMap [102].

O ConfigMap é um mecanismo do Kubernetes que permite a injeção de configurações em contentores. Ao criarmos um ConfigMap especificamos um conjunto de ficheiros ou pares chave-valor que são carregados para o armazenamento persistente do cluster Kubernetes, o etcd. No momento de *deployment* de um Pod invocamos o ConfigMap e especificamos o local dentro do contentor (por exemplo um diretório) onde os ficheiros de configuração devem constar para que possam ser lidos.

Lidar com Informação Sensível (*Secrets*)

Da mesma forma que no caso da gestão de configurações falado anteriormente os dados sensíveis como por exemplo chaves ou *passwords* não devem constar do conteúdo da imagem. Não só porque variam de ambiente para ambiente mas também por causa da sensibilidade dos dados em si. Para injetar este tipo de informação no *Pod* o Kubernetes utiliza o objeto *Secret* [103].

Auto-Healing de Contentores

Outra funcionalidade que traz benefícios às aplicações é o sistema de *auto-healing* do Kubernetes. Este sistema permite que quando uma aplicação entra num estado inconstante ou problemático e o seu processo morre, o Kubernetes trata de relançar esse Pod, possivelmente noutro nó do *cluster*. Isto faz com que a resiliência da aplicação aumente.

Aplicações *Stateful* e Persistência dos Dados

Um grande desafio ao *deployment* de aplicações conteinerizadas em *clusters* é o caso das aplicações com estado (*stateful*). Garantir persistência de dados num ambiente em *cluster* em que os contentores podem percorrer vários nós é uma tarefa complexa. Para resolver esta situação o Kubernetes criou o conceito de *Persistent Volumes* [104].

Os *Persistent Volumes* são recursos com um ciclo de vida independente do ciclo de vida dos *Pods*. Num ambiente *cloud* em que o *cluster* Kubernetes é suportado por um orquestrador de IaaS, como por exemplo o OpenStack, os *Persistent Volumes* são um recurso da *cloud* e externo

ao *cluster* Kubernetes (por exemplo os volumes Cinder do OpenStack podem ser *Persistent Volumes*). Neste cenário quando um *Pod* morre e é relançado noutro nó o *Persistent Volume* é montado pelo OpenStack nesse novo nó, para que o *Pod* possa aceder a todos os dados que escreveu até ao momento da sua morte. Este movimento de volumes é realizado pela interação do Kubernetes com o orquestrador de *cloud*.

Quando o cluster é criado sobre um ambiente *bare-metal* pode ser utilizado volumes externos disponibilizados pela rede, por exemplo NFS.

Atualizações Sem Quebras de Serviço

A realização de atualizações aos objetos Pod que pertencem ao objeto Deployment é realizada sem quebras de serviço. Isto acontece devido a um processo automático em que as atualizações aos Pods é realizada incrementalmente, ou seja, Pod a Pod [105]. Apenas quando o *deployment* dos Pods com a nova versão atualizada é concluído com sucesso é que o tráfego dos pedidos é redirecionado para os novos Pods e os Pods da versão antiga são eliminados.

3.4.3.3 Deployment de Aplicações em Kubernetes

As aplicações conteinerizadas através da plataforma Docker realizam o seu *deployment* no ambiente Kubernetes através de *templates* escritos em YAML. Na sua forma mais simples um Deployment pode ser escrito da seguinte forma:

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Neste exemplo é realizado o Deployment com o nome `nginx-deployment` indicado pelo campo `metadata: name`, ao qual são criadas 3 réplicas (campo `spec: replicas:`) de Pods com a imagem `nginx` na versão 1.7.9 (campo `spec: containers: image:`). Este *deployment* abre a porta 80 para ser utilizada pelos Pods.

Estes templates YAML podem configurar todos os objetos do Kubernetes. Por exemplo, para os Pods criados pelo Deployment anterior poderem ser acedidos externamente e esses pedidos

serem balanceados pelas 3 réplicas é necessário criar o objeto Service, como se pode ver pelo exemplo a seguir:

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: nginx
  type: NodePort
  ports:
  - protocol: TCP
    port: 80
    targetPort: 30380
```

No exemplo anterior é criado um Service com o nome `my-service` que irá balancear o tráfego por todos os Pods com as Labels `app: nginx`. Como este Service é do tipo NodePort será exposto externamente ao cluster através da porta 30380 e os pedidos que chegarem a esta porta serão balanceados em *Round Robin* pelos os Pods que estão na alçada deste Service.

3.4.3.4 Arquitetura

O Kubernetes é composto por vários elementos com funcionalidades ao nível da administração e execução de contentores que cooperam entre si formando o *cluster*. Estes elementos são organizados logicamente por dois tipos de nós *Master* e *Worker* de acordo com as suas funcionalidades, como se pode ver pela Figura 3.5. Nesta secção vamos abordar cada um desses elementos.

Etcd

Etcd é uma base de dados distribuída que armazena pares chave valor e fornece uma forma de armazenar informação consistente num *cluster* de máquinas [107]. Nesta base de dados o Kubernetes armazena toda a informação relativa ao *cluster* permitindo por exemplo a descoberta de serviços ou a partilha de configurações.

API Server

Este é o elemento responsável por expor a interface REST através da qual todas as interações relacionadas com os objetos do Kubernetes são validadas e efetuadas [108].

controller-manager

O controller-manager como o próprio nome indica tem a tarefa de controlar. Para isso corre um conjunto de controladores individuais que realizam tarefas em rotina, em que as mais relevantes são:

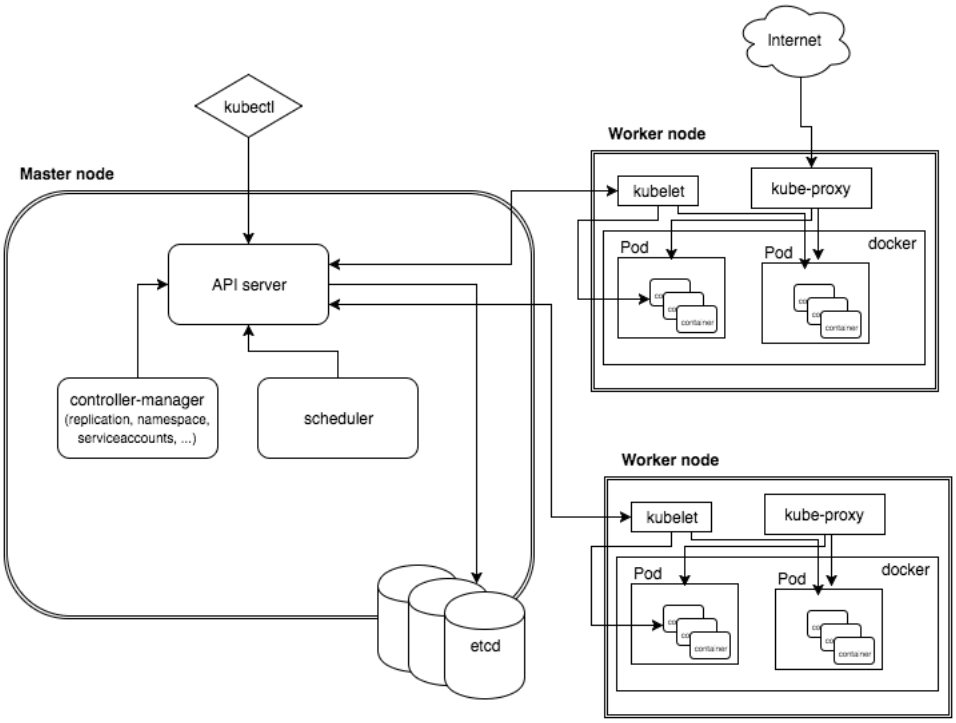


Figura 3.5: Arquitetura do Kubernetes (Fígura extraída de [106]).

- **Node Controller:** Responsável por observar e reagir aos eventos de estado dos nós do *cluster*;
- **Replication Controller:** Responsável por manter o número de réplicas correto de cada Pod;
- **Service Account e Token Controllers:** Responsável por lidar com questões de autenticação e autorização;

scheduler

Este elemento é responsável por selecionar qual o nó que preenche os requisitos de um Pod a ser lançado.

kubelet

O kubelet é o elemento primário do nó *Worker* que recebe instruções do kube-apiserver e a sua principal tarefa é instruir o Docker para operar os contentores dos Pods alocados à *host* e observar o seu estado através de *probes*. Para além disso, este tem a tarefa de montar os volumes necessários ao Pod e de reportar o estado do seu nó e dos seus Pods ao resto do sistema.

kube-proxy

O kube-proxy é o elemento responsável por criar a abstração fornecida pelos Services falados anteriormente. Para isso este elemento cria e gere um conjunto de regras de rede na *host* permitindo o redirecionamento e balanceamento dos pedidos pelos vários Pods.

kubectl

Kubectl é a interface de linha de comandos que executa comandos no *cluster* Kubernetes [109].

3.4.3.5 Rede

O Kubernetes garante que os Pods podem comunicar uns com os outros independentemente do nó do *cluster* onde se encontrem. Para que isto aconteça é fornecido a cada Pod um endereço IP interno ao *cluster* que garante comunicação entre os Pods sem necessidade de criar *links* explícitos entre Pods ou mapeamento entre as portas dos contentores e as portas da *host*. Do ponto de vista da alocação de portas, dos nomes, da descoberta de serviços, do balanceamento de carga, da configuração de aplicações e migração os Pods podem ser tratados como máquinas virtuais ou *hosts* físicas [110].

Este modelo de rede satisfaz os seguintes requisitos de rede:

- Todos os contentores podem comunicar com todos os outros contentores;
- Todos os nós do *cluster* podem comunicar com todos os contentores (e vice-versa);
- O endereço IP que um contentor vê (por exemplo pelo comando ifconfig) é o mesmo endereço IP que os outros contentores vêem;

Implementação da Rede - Flannel

Para implementar o modelo de rede que satisfaz os requisitos de comunicação do Kubernetes pode ser utilizado o Flannel [111]. Flannel é um sistema simples capaz de implementar uma rede de *overlay* na camada de rede (*layer* 3) que aloca uma *subnet* de um espaço de endereçamento pré configurado a cada nó do *cluster* transportando o tráfego entre os nós.

Capítulo 4

Execução e Gestão de Aplicações Conteinerizadas

Neste capítulo vamos identificar as estratégias para lidar com aplicações conteinerizadas desde o desenvolvimento até à fase de produção. Será traçado um caminho que mostra o processo de desenvolvimento de aplicações conteinerizadas com os seus vários componentes que são capazes de lidar com a aplicação nas várias fases do seu ciclo de vida. Posteriormente serão levantados os fatores chaves para que possam ser abordados e analisados com mais detalhe nas secções subsequentes. Esses fatores chaves focam-se essencialmente no desenvolvimento, validação e distribuição de imagens, no *deployment* de aplicações conteinerizadas sobre um cluster Kubernetes.

4.1 Do Desenvolvimento à Produção

Nesta secção vamos identificar as etapas necessárias que uma aplicação necessita de transitar para que seja conteinerizada, desde o ambiente de desenvolvimento até à entrada no ambiente de produção. Para isso, foi criado um *workflow* dividido em várias fases onde é possível ter uma visão global do itinerário que uma aplicação tem de percorrer. Desta forma espera-se determinar os componentes e sistemas necessários para criar uma *pipeline* de desenvolvimento e produção de aplicações em contentores.

Tendo em conta as práticas automáticas de desenvolvimento de *software* abordadas na secção 2.5 nomeadamente CI e CD foi possível criar o seguinte *workflow* de desenvolvimento de *software* esquematizado na Figura 4.1.

O diagrama para o desenvolvimento de aplicações conteinerizadas foi dividido em quatro fases essenciais, que são as fases de *Create*, *Build*, *Deploy* e *Run*.

Fase *Create*

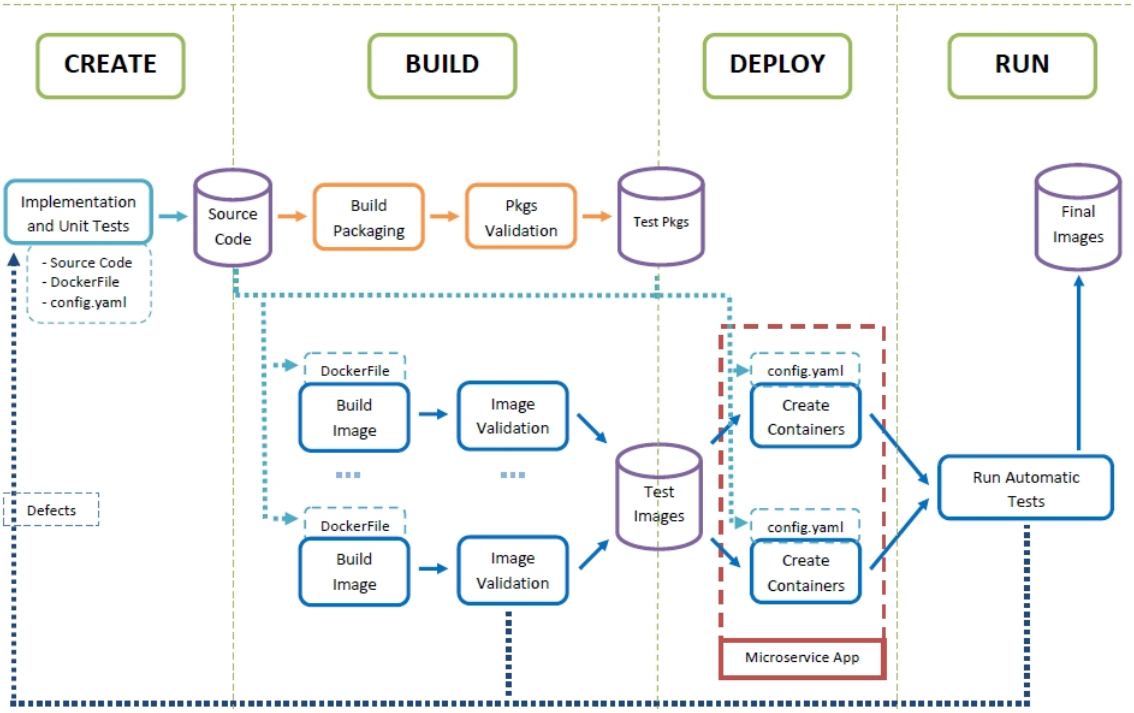


Figura 4.1: *Workflow* proposto para o desenvolvimento de aplicações conteinerizadas.

Esta fase é onde se encontra a componente humana composta pelas equipas de trabalho que desenvolvem o *source code*. Para além do *source code* as equipas terão de criar Dockerfiles (abordadas na secção 3.3) de maneira a automatizar o processo de construção de imagens. Para que estas imagens possam ser instanciadas em contentores de maneira a executar os serviços da aplicação deve ser criado um conjunto de ficheiros de configuração YAML (conceito abordado na Secção 3.4.3) que vão permitir a realização do *deployment* da aplicação no *cluster* Kubernetes. Tanto as Dockerfiles como os ficheiros de configuração YAML vão ser colocados no repositório central de *source code*.

Depois da fase *Create* as fases seguintes vão ser constituídas por processos automáticos que vão permitir levar a aplicação até ao ambiente de produção em contacto com o cliente.

Fase *Build*

A fase de Build foi criada para que pudesse utilizar *software* empacotado em pacotes RPM. Com isto espera-se que o *software* que já esteja a ser desenvolvido e empacotado no formato de RPM possa ser mais facilmente conteinerizado.

Nesta fase não é a construção de RPMs que merece mais atenção mas sim a criação de imagens através das Dockerfiles. Após as imagens estarem criadas, estas têm de ser validadas de maneira a perceber se estão de acordo com um conjunto de regras. Após a validação de todas as regras as imagens podem ser enviadas para o repositório de testes de maneira a dar inicio à fase seguinte de realização do *deployment* da aplicação no *cluster* Kubernetes.

Fase *Deploy*

Nesta fase as imagens serão instanciadas em contentores e implantadas num *cluster* Kubernetes o mais semelhante possível com o ambiente de produção, para que na próxima fase se possa proceder à realização dos testes.

Fase *Run*

Por fim surge a fase de *Run* que vai ser o ambiente em que a aplicação está em execução e totalmente funcional sobre o *cluster* de testes para que possam ser iniciados todos os testes à aplicação.

4.2 Fatores Chave

Adotar esta abordagem por parte de uma organização é uma tarefa que necessita de ser organizada e ter um elevado rigor técnico. Considerando o diagrama lógico apresentado na Figura 4.1 existe um conjunto de tópicos que devem ser considerados com mais atenção, que são:

- **Desenvolvimento de Imagens:** O processo de desenvolvimento de imagens. Será abordado na Secção 4.3;
- **Validação de Imagens:** O processo e o mecanismo de validação de imagens. Será abordado na Secção 4.4;
- **Distribuição de Imagens:** O tipo de plataforma que vai armazenar e distribuir as imagens. Será abordado na Secção 4.5;
- **Orquestração de Contentores:** O ambiente que vai suportar a aplicação conteinerizada. Será abordado na Secção 4.6;

Nas secções seguintes iremos abordar as soluções técnicas existentes para cada um destes tópicos listados anteriormente.

4.3 Desenvolvimento de Imagens

As imagens Docker são consideradas uma nova forma de empacotamento e distribuição de *software* pronto a ser executado, como foi abordado na Secção 3.3. Estas imagens estão disponíveis em repositórios públicos, em que o mais conhecido é o Docker Hub [76]. A partir do Docker Hub podemos descarregar vários *softwares* desde servidores *web* (por exemplo Nginx) ou até SGBD (por exemplo PostgreSQL), em que a maior parte das vezes estas imagens são oficiais e por isso são criadas e disponibilizadas pelas entidades que as desenvolvem. Mas para algumas organizações esta forma de obter imagens ainda levanta algumas questões ao nível da segurança e do seu conteúdo motivando a que criem as suas próprias imagens oficiais. Independentemente disso, para uma organização que pretende utilizar conteinerização e distribuir o seu *software* em imagens é crucial ter o *know-how* suficiente do processo de construção de imagens.

Empacotamento de *Software* em Imagens

Um programador pode criar imagens diretamente de um contentor. Para isso basta tirar um *snapshot* ao contentor com o *software* a distribuir instalado e passamos a ter uma imagem com esse *software*. Isto é possível de ser realizado porque os contentores utilizam um sistema de ficheiros de união (*Union File System* (UFS)) que permite que qualquer alteração realizada ao contentor seja criada numa nova camada, como falado na Secção 3.3.

O *workflow* básico para construir uma imagem a partir de um contentor é o seguinte:

- Em primeiro lugar é necessário arrancar um contentor a partir de uma imagem existente. Por exemplo uma imagem base de um sistema operativo, como por exemplo CentOS.
- Em segundo lugar é necessário modificar o sistema de ficheiros do contentor em execução, para que estas alterações sejam registadas na pilha de camadas, como foi falado na secção 3.3. Por exemplo instalando e configurando um *software* dentro do contentor.
- Por fim quando todas as alterações forem efetuadas com sucesso basta fazer `docker commit` das alterações e passamos a ter uma nova imagem [112]. Esta nova imagem pode ser utilizada como qualquer outra imagem.

Este processo é funcional mas não é eficaz pois não permite automatizar e repetir a construção da imagem, realizar facilmente alterações ou atualizações ao *software* e torna a tarefa de auditar o conteúdo de uma imagem muito mais complexo.

Em alternativa a este método podemos utilizar um método documentável e sustentável que permite automatizar o processo de construção de imagens. Este método utiliza os ficheiros de configuração chamados Dockerfiles (abordadas na Secção 3.3) que permite declarar todas as instruções e passos que vão possibilitar a criação da imagem. Ao utilizarmos este método já não temos que lidar com os problemas do método anterior e tirámos partido de todas as vantagens do Docker como, por exemplo, a reutilização de camadas.

Eficiência e Organização em Camadas

Uma aplicação normalmente é dividida em várias imagens pelo menos uma por cada serviço. Cada uma destas imagens tem um padrão de desenvolvimento comum em que o ponto de partida (`FROM:`) é uma imagem base com um sistema operativo e depois em cima desta são instalados os *softwares* e dependências que suportam todos os serviços da aplicação. De seguida, sobre esta camada de dependências é instalado o *software* específico ao serviço. Tendo em conta isto, a construção das imagens pode ser organizada em duas camadas. A primeira é a construção da camada *framework* e a segunda é a construção da camada aplicação, como se pode ver pela Figura 4.2.

A camada *framework* normalmente vai ser composta pelas bibliotecas do sistema operativo e por todos os *softwares* que são comuns a todos os serviços da aplicação. Enquanto que a camada da aplicação vai ser composta apenas pelo *software* específico ao serviço.

Esta criação de imagens organizada em duas camadas permite aumentar a agilidade e o controlo no *deployment* e na realização de *updates*. No primeiro caso como a imagem *framework* é

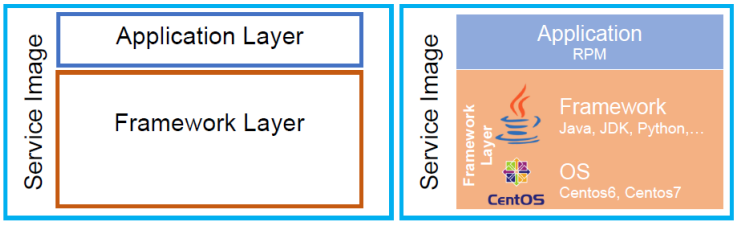


Figura 4.2: Organização do desenvolvimento de imagens de serviços em duas camadas.

partilhada por todos os outros serviços garante que o ponto de partida é o mesmo para todos. Por exemplo, permite que todos partilhem os mesmos requisitos ou controlos ao nível da segurança. Para além disso, a utilização de duas camadas permite que a realização de *updates* que são comuns a todos os serviços da aplicação possam ser realizadas na camada *framework* não implicando alterações na camada aplicação. E *updates* específicos ao serviço possam ser realizados na camada aplicação.

Com esta abordagem aumentamos a organização e o controlo na construção das imagens que compõe todos os serviços da aplicação.

Imagens Base: Públicas ou Privadas

Como vimos a construção de qualquer imagem de um *software* tem como ponto de partida uma imagem pai ou base, que normalmente é a imagem de um sistema operativo. Então para que seja possível a construção de imagens é necessário obter externamente esta imagem base inicial ou criá-la internamente na organização.

Existem várias imagens oficiais disponíveis no Registry público Docker Hub para CentOS, Ubuntu, etc. Ou então versões simples e reduzidas em tamanho como por exemplo Busybox, Alpine, etc. Para além das imagens baseadas em sistemas operativos existem imagens públicas já com *frameworks* ou aplicações instaladas como por exemplo Nginx, Postgres, Node, etc. Com isto concluímos que existe uma panóplia de imagens diferentes que podem ser usadas como pontos de partida para outras imagens ou então imagens prontas a ser executadas, que podem ser personalizadas.

Numa organização a utilização de imagens públicas pode trazer alguns problemas. O problema mais evidente é a dependência externa noutra organização. O estudo realizado por Shu et al. [113], que avaliou cerca de 300.000 imagens, concluiu que 50% das imagens públicas não eram atualizadas por mais de 200 dias e outros 30% das imagens não eram atualizas há mais de 400 dias. Outro dado retirado deste estudo que é ainda mais alarmante é que em média as imagens consideradas tanto oficiais como comunitárias continham mais de 180 vulnerabilidades diferentes e as imagens filho que derivavam destas acrescentavam em média mais 20 vulnerabilidades ao nível da segurança.

Visto que a adoção de imagens públicas faz com que uma organização perca o controlo da sua base de desenvolvimento e abre um grande número de vulnerabilidades nos seus *softwares* a

melhor opção a considerar é a construção de imagens internas e privadas.

Uma imagem base deve ser o mais leve possível pois torna mais rápido o *deployment* da aplicação. Uma imagem pode começar totalmente do zero (a que é chamado *scratch*) e apenas conter um executável sem dependências. Este modo não é comum de ocorrer pois normalmente estas precisam de certas dependências ou *runtimes* consoante a linguagem em que o *software* é escrito, por exemplo Java, Python, etc. Para a instalação destas dependências em contentores são necessárias ferramentas que são inerentes aos sistemas operativos, por exemplo o YUM no caso da instalação de RPMs ou então o apt-get no caso da instalação e empacotamento dos sistemas GNU/Linux Debian, entre outros. Para cada um dos casos é necessário criar uma imagem base contendo estas dependências do sistema operativo a ser utilizado.

Tendo em conta a informação encontrada na documentação oficial do Docker para a construção de imagens bases [114], a construção de uma imagem base do CentOS é feita a partir de uma máquina com esse sistema operativo executando com o Docker um *script*, disponível em [115]. Para a construção de uma imagem baseada em Ubuntu é utilizada a ferramenta "*debootstrap*" que permite instalar um sistema baseado em Debian num subdiretório de outro sistema. Outro exemplo da criação de uma imagem que tem um tamanho muito reduzido é a Alpine baseada em BusyBox cuja a construção pode ser feita à custa de um Dockerfile disponível em [116].

Processo Automático para Construção de Imagens

Após estar definida qual a imagem base e os programadores terem desenvolvido as linhas iniciais do *source code* para cada serviço chega a fase de empacotamento e criação de imagens. Como falado na Secção 3.3, as imagens são criadas à custa de uma Dockerfile que é escrita pelos programadores e que estará no mesmo repositório que o *source code*. O programador antes de realizar *commit* para o repositório central testa todas as alterações no seu ambiente local tanto ao nível do *source code* como ao nível da construção da imagem.

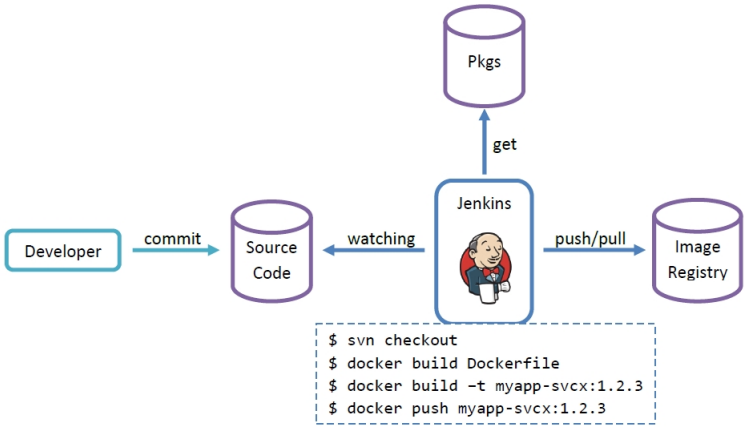


Figura 4.3: Processo automático para construção de imagens.

Como pode ser visto pelo diagrama da Figura 4.3, o *workflow* para a construção de imagens é iniciado sempre que é feito uma alteração ao código ou então em intervalos de tempo regulares, desencadeando as seguintes ações no servidor Jenkins de CI (abordado na Secção 2.5):

1. É feito *check out* do repositório para que o servidor tenha a última versão do código;
2. É feito o *build* da imagem à custa da Dockerfile, que pode incluir a instalação de RPMs;
3. Podem ser realizadas um conjunto de validações que dependem da aplicação ou serviço;
4. A imagem é etiquetada com a versão e é enviada para o repositório de imagens passando a estar disponível;

4.4 Validação de Imagens

Validar uma imagem é um processo necessário para que uma organização tenha um controlo sobre a qualidade das suas criações. Para além disso, a validação permite assegurar que as imagens foram criadas por um processo específico que é confiável e não por outro processo qualquer, que por exemplo extrai *software* malicioso da Internet. Por outro lado, a existência de um processo de validação de imagens que consecutivamente valida trabalho faz com que a exigência das pessoas sobre o seu trabalho aumente o que é algo muito positivo para uma organização.

Os pontos que devem ser validados numa imagem variam muito de organização para organização, pois podem ter em conta vários fatores inerentes à organização. Mas no mínimo deve-se garantir os seguintes pontos:

1. O nome e versão da imagem estão de acordo com a versão do *software* no repositório central;
2. A imagem tem todos os metadados necessários;
3. A imagem base ou pai utilizada é de origem interna e encontra-se devidamente validada;
4. No decorrer do processo de *build* não existe nenhuma ligação à rede externa (Internet);
5. O *source code* incluído na imagem encontra-se no repositório central;

Os pontos anteriormente enumerados podem ser alcançados de algumas formas, como vamos ver a seguir.

Histórico

Uma imagem quando criada através de Dockerfiles não se torna numa caixa negra que ninguém consegue ver e auditar o seu conteúdo. O Docker fornece o comando *history* que permite ver as várias camadas que compõem uma imagem com informação adicional do comando ou ação que criou essa camada, como podemos ver pelo excerto da Figura 4.4. Recordemos que um conjunto de camadas formam uma imagem, como foi falado na Secção 3.3.

Conteúdo

Uma imagem pode ser convertida num ficheiro TAR o que permite ter acesso a todas as camadas e consecutivamente a todos os diretórios e ficheiros que compõe cada uma destas camadas. Desta forma podemos proceder a uma validação mais especifica, num nível mais baixo, analisando diretório a diretório, ficheiro a ficheiro.

Para fazer isto basta recorrer aos comandos:

```
[dreis@localhost ~]$ docker history d0b74141fcda
IMAGE               CREATED             CREATED BY                                      SIZE
d0b74141fcda        2 weeks ago         /bin/sh -c #(nop)  CMD ["-c url_conf"]          0 B
<missing>           2 weeks ago         /bin/sh -c #(nop)  ENTRYPOINT ["/opt/ptin/sma   0 B
<missing>           2 weeks ago         /bin/sh -c #(nop)  EXPOSE 3000/tcp              0 B
<missing>           2 weeks ago         /bin/sh -c yum -y install /data/smartiot-http   209.4 MB
<missing>           2 weeks ago         /bin/sh -c yum -y install env-jruby2 && yum i   42.14 MB
<missing>           2 weeks ago         /bin/sh -c #(nop)  CMD ["/bin/bash"]            0 B
<missing>           2 weeks ago         /bin/sh -c yum -y install env-java1.8.noarch    404.9 MB
<missing>           9 weeks ago                                                         183.1 MB
```

Figura 4.4: *Output* do comando docker history.

```
$ docker save <IMG_ID> -o output.tar
$ tar -xvf output.tar
```

Criar de Forma Segura

Todo o processo de validação de imagens torna-se mais simples se a criação das imagens for realizada de forma segura. Criar as imagens com base num processo manual em que o programador no seu ambiente de desenvolvimento faz o *build* e *upload* diretamente para o repositório de imagens é algo que não é aconselhável porque pode levar a erros de criação ao nível dos metadados ou pode ser incluído *software* externo malicioso. Visto que a criação das imagens deve ser realizada por um processo automático e repetitivo à custa por exemplo de um servidor de integração contínua torna todo o processo de validação eficaz e simples.

Se o servidor que vai criar as imagens for colocado num ambiente que não tem ligação à rede externa e apenas tem ligação aos repositórios *source-code*, pacotes e imagens internas temos logo garantia que os pontos 3, 4, 5 (apresentados anteriormente) são garantidos pois caso contrário o processo de *build* falha. O ponto 1 é garantido pelo servidor de integração contínua pois é ele que vai integrar o *source code* de uma versão especifica na imagem e essa versão será atribuída à imagem. Quanto ao ponto 2 que diz respeito à validação dos metadados, isto pode ser realizado após o *build* através do comando `docker inspect` que lista todos os metadados de uma imagem.

4.5 Distribuição de Imagens

Após a criação das imagens estas devem ser disponibilizadas nalgum local para que possam ser utilizadas tanto pelas equipas de trabalho como pelos clientes. Existem algumas alternativas para a distribuição das imagens que dependem do nível de restrições ou da capacidade de infraestrutura da organização.

Existem 4 formas diferentes para distribuir imagens que são: repositórios externos públicos, repositórios externos privados, repositórios internos e repositórios de ficheiros. As várias opções para a criação de repositórios estão organizadas de acordo com a sua simplicidade e flexibilidade, que pode ser vista no diagrama da Figura 4.5. De seguida vamos explorar cada uma das possibilidades existentes para a distribuição de imagens apresentada.

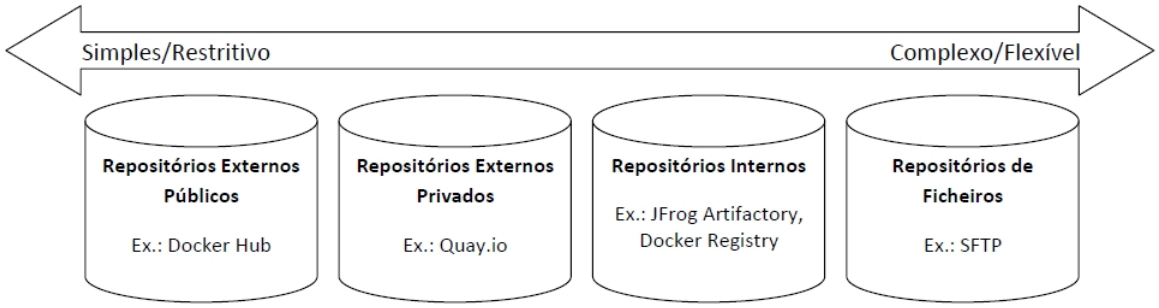


Figura 4.5: Espectro de possibilidades para distribuição de imagens. (Imagem adaptada de [22])

Repositórios Externos Públicos

Os repositórios externos públicos como o próprio nome indica são repositórios que são acedidos por qualquer entidade através da Internet e são mantidos por uma organização externa. Normalmente nestes repositórios encontramos imagens públicas em que o seu conteúdo é *open source*. Os exemplos mais conhecidos destes repositórios são o Docker Hub e o Google Container Registry.

Estes repositórios normalmente são grátis, com uma grande visibilidade devido à sua popularidade, mas têm um fraco controlo dos acessos às imagens disponibilizadas, como foi dito qualquer pessoa pode transferir estas imagens. Para uma organização que distribui *software* empacotado em imagens e que retire lucros da utilização deste *software* esta não é a melhor opção.

Repositórios Externos Privados

Este tipo de repositórios é muito semelhante aos repositórios anteriores do ponto de vista operacional. A diferença é que neste caso é necessária uma autenticação para aceder às imagens, daí serem considerados privados.

Estes repositórios normalmente têm custos associados que podem aumentar consoante o número de repositórios necessários. Para organizações de grandes dimensões esta não será a melhor opção pois necessitam de uma grande quantidade de repositórios o que leva a custos excessivos. Para além disso, esta opção pode tornar a organização dependente de outra organização externa.

Repositórios Internos

Para uma organização que necessita de controlar todos os aspetos de um repositório como os acessos ou confidencialidade das imagens a utilização de repositórios internos é a mais aconselhada. Estes repositórios ficam na alçada da empresa que tem controlo sobre todo o repositório e sua infraestrutura.

A opção mais conhecida para a criação de repositórios internos privados é o Docker Private Registry [77]. Esta opção é *open source* e é disponibilizada através de uma imagem Docker. O processo de instalação do repositório é simples, pois basta lançar um contentor com a imagem

do Docker Registry a partir do Docker Hub [76] que passamos a ter um repositório interno e privado.

Para além do Docker Registry existem outras opções como por exemplo JFrog Artifactory [117] que fornece suporte, mas com custos associados.

Repositórios de Ficheiros

Uma imagem é um ficheiro o que faz com que estas possam ser distribuídas como qualquer outro ficheiro, por exemplo através de um sistema de armazenamento comum disponibilizado pela rede. Este tipo de distribuição como não tem a noção do conceito de imagem tem uma interoperabilidade fraca com as ferramentas Docker, fazendo com que não tire partido do conceito de reutilização de camadas ou do versionamento de imagens.

A maior utilidade deste tipo de repositórios é a distribuição de imagens em espaços que não têm acesso à Internet e estão isolados. Para que as imagens se tornem ficheiros para que possam ser exportadas é utilizado o comando `docker save -o file.tar` em que o seu *output* é um ficheiro TAR que pode ser exportado para um servidor SFTP, Web, Email ou até um dispositivo de armazenamento USB.

4.6 Orquestração de Contentores

Com a criação de *softwares* de larga escala, com arquiteturas baseadas em micro-serviços com funcionalidades ao nível da *Internet of Things* (IoT) o número de contentores dedicados a uma aplicação chega facilmente às dezenas ou por vezes centenas. Gerir todos estes contentores manualmente num *cluster* de máquinas é uma tarefa impossível. Devido a isso é necessário uma ferramenta extra que consiga criar a camada de orquestração. É nesta camada que entra as plataformas de gestão de contentores em *cluster* abordadas na Secção 3.4.3.

A ferramenta selecionada que vai permitir a gestão de contentores em *cluster* foi o Kubernetes. Ambas as soluções abordadas na Secção 3.4.3 tem vantagens e desvantagens e a sua utilização depende das características e dos produtos de *software* da organização. Nesta dissertação a escolha do Kubernetes deve-se às seguintes razões:

- O Kubernetes é a ferramenta mais popular entre os seus pares e como este projeto é *open-source* traz uma maior confiança ao desenvolvimento e continuidade do projeto [118];
- O *deployment* em ambiente Kubernetes de vários sistemas como por exemplo Zookeeper, Kafka, Apache Spark e muitos outros estão documentados na *web* o que facilita a criação dos *deployments* internos de qualquer organização;
- É uma ferramenta com uma gestão de configurações bastante completa que tem objetos que ajudam o desenvolvimento de aplicações conteinerizadas como é o caso dos ConfigMaps, Secrets, StatefulSets.
- Esta ferramenta é totalmente integrável com o sistema OpenStack o que permite um maior controlo sobre o *cluster* Kubernetes, quando este é criado sobre nós virtuais.

- A alocação de volumes externos persistentes é realizada pelo Kubernetes ao comunicar com o OpenStack o que permite a criação de aplicações com estado com uma melhor performance, sem a instalação de ferramentas adicionais.
- A plataforma Kubernetes é bastante completa sem existir necessidade de instalar ferramentas extras para o seu funcionamento.

Nesta secção vamos abordar o Kubernetes começando por perceber as formas de instalação e configuração do *cluster*. De seguida vamos apresentar o *cluster* mínimo criado para produção com requisitos de alta disponibilidade. Após isso e para aumentar as capacidades do *cluster* este será estendido aos conceitos de escalabilidade automática com a criação de um comando que permita lançar grupos de nós elásticos.

4.6.1 Instalação do Cluster

Para podermos realizar o *deployment* e a execução de contentores no *cluster* Kubernetes é evidentemente necessário criar o *cluster*. Existem três tipos de *cluster* Kubernetes que são normalmente necessários numa organização. Um de teste local que é composto por apenas um nó e serve para que o programador faça pequenos testes aos ficheiros YAML para *deployment* dos serviços. O *cluster* de testes com características iguais ao *cluster* de produção, que serve para testar o *deployment* e a execução total de uma aplicação com todos os serviços antes de ir para produção. E por fim, existe o *cluster* de produção que é este que está a fornecer o serviço ao cliente e por isso tem de ser o mais robusto possível.

Instalação Manual

Uma das primeiras alternativas para a criação de um *cluster* Kubernetes é instalar e configurar este manualmente máquina a máquina, serviço a serviço. Esta alternativa é bastante demorada e não permite repetições o que leva a que sempre que seja necessário criar um novo *cluster* tenha-se de repetir todo o processo manual novamente. Para além disso, para realizar alterações ou atualizações ao *cluster* é necessário mais uma vez um processo individual, máquina a máquina, o que por vezes pode levar a ligeiras diferenças entre os nós que compõe o *cluster* ou diferenças entre o ambiente de testes e o ambiente de produção. Devido a estes fatores esta alternativa não é uma boa opção sendo necessária a utilização de processos automáticos e repetitivos.

Uma das vantagens da criação de um *cluster* por processos manuais é a possibilidade de criar um *cluster* mais adaptado às infraestruturas e necessidades da organização. [119] Outra vantagem que foi sentida no desenrolar desta dissertação foi a introdução de conhecimento nas equipas de trabalho. Pois como este processo é bastante manual e requer que todos os serviços que compõe o *cluster* Kubernetes sejam instalados e configurados individualmente, faz com que se explore e aprenda os conceitos arquiteturais do Kubernetes de uma maneira mais prática.

Instalação Automática (Ansible)

Esta alternativa é uma evolução da alternativa anterior que vem corrigir alguns defeitos. O

Ansible é uma ferramenta que permite automatizar a instalação e configuração de *softwares* a partir de um ponto central. [120] Então, após as equipas conhecerem todos os detalhes do *cluster* podem criar *playbooks* Ansible que descrevem todos os *softwares* que devem ser instalados assim como as suas configurações. Após isto, basta correr esses *playbooks* sobre um grupo de máquinas definido que a instalação será concluída automaticamente.

Existe alguns *playbooks* que o Kubernetes sugere como é o caso do KubeSpray [121] que é agnóstico tanto ao sistema operativo das *hosts* como ao ambiente *cloud* utilizado.

Num ambiente *bare-metal* existe outra opção recomendada pelo Kubernetes que pode ser obtida em [122, 123]. Para máquinas com sistema operativo CentOS existe um projeto público no Github [124] que pode servir como ponto de partida para uma instalação de um *cluster* Kubernetes em alta disponibilidade.

Instalação no Ambiente Local

Para criar um ambiente de testes local em que na maior parte das vezes é instalado na máquina de desenvolvimento do programador existe a opção Minikube [125]. Esta opção cria um ambiente Kubernetes com apenas um nó e pode ser instalado diretamente sobre *hypervisor* VirtualBox o que permite a instalação nos sistemas operativos mais populares [126].

4.6.2 Cluster Mínimo

O *cluster* Kubernetes é composto por vários serviços e sistemas com funcionalidades ao nível de *Master* ou de *Worker*, como foi visto na Secção 3.4.3. Na sua arquitetura lógica o *cluster* é dividido em *Master Nodes* e em *Worker Nodes*. Mas para uma reutilização e eficiência ao nível dos recursos físicos estes dois tipos de *Nodes* podem ser aglomerados num só nó, como é ilustrado na Figura 4.6.

Número de Elementos do *Cluster*

Para termos um *cluster* totalmente funcional com alta disponibilidade e persistência é necessário no mínimo 3 máquinas físicas ou virtuais. Esta dimensão do *cluster* é devido à base de dados etcd que é um dos elementos essências da arquitetura do cluster Kubernetes como foi visto na Secção 3.4.3.

O etcd é uma base de dados que armazena pares chave-valor de forma consistente e em alta disponibilidade e que permite ao Kubernetes armazenar com persistência todos os objetos da API [127]. O etcd funciona em *cluster* onde existe um líder e n seguidores. O processo de eleição é feito à custa do algoritmo de Raft [128]. Neste algoritmo um nó passa a ser líder se receber uma maioria de votos dos outros nós. Por exemplo num *cluster* com 5 elementos um líder ganha eleição se receber pelo menos 3 votos, ao qual um destes votos foi efetuado pelo próprio nó que está a ser eleito. Isto faz com que 2 elementos do *cluster* possam falhar que continua a existir maioria no processo de eleição. Se imaginarmos um cenário com 6 elementos, ou seja, um cenário par, a maioria é obtida com 4 votos e continuamos a ter 2 elementos que podem falhar. Neste

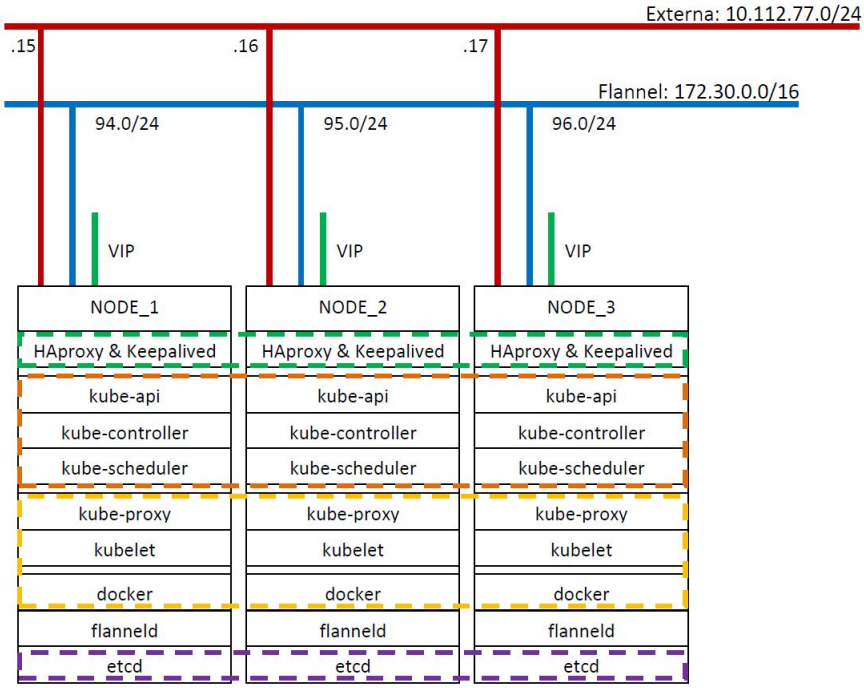


Figura 4.6: Cluster Kubernetes mínimo e a distribuição dos seus serviços pelos 3 nós dos *cluster*.

Tabela 4.1: Comparação de um *cluster* etcd com um número de elementos par e ímpar. (Tabela adaptada de [129])

Tamanho do *Cluster*	Maioria	Tolerância à Falha
1	1	Nenhum
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3
8	5	3
9	5	4

último cenário acrescentamos mais um nó ao *cluster* mas não ganhamos tolerância a falha pois tanto no cenário com 5 ou 6 elementos apenas podem falhar dois nós para continuar a existir maioria no processo de eleição. Na Tabela 4.1 são comparados cenários em que o número de elementos do *cluster* é par ou ímpar e onde é possível ver a quantidade de elementos para obter maioria assim como a quantidade de elementos que podem falhar face ao número de elementos do *cluster*.

Cluster em Alta Disponibilidade e com Balanceamento de Carga

Para permitir efetuar o balanceamento dos pedidos externos (realizados pelos clientes) pelos nós do *cluster* é necessário instalar e configurar um balanceador de carga, como pode ser visto pelo diagrama da Figura 4.7.

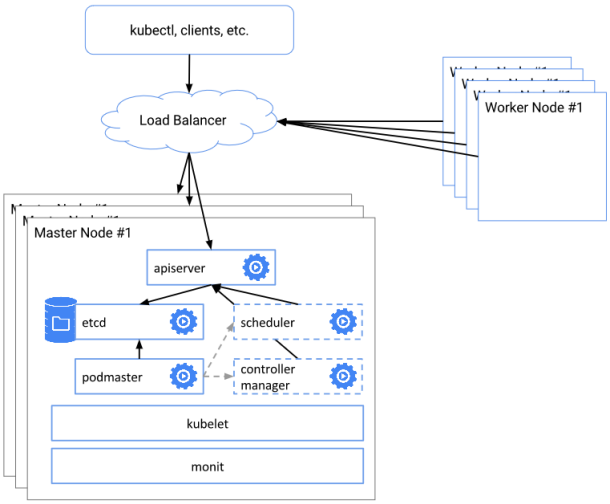


Figura 4.7: *Cluster* Kubernetes em Alta Disponibilidade (Figura extraída de [130]).

Em ambientes *cloud* que disponibilizam o serviço de balanceamento de carga este pode ser automaticamente provisionado pela própria *cloud* e não é instalado nos nós do *cluster* Kubernetes, como por exemplo o caso da Google Cloud Plataform [131]. Quando não existe esta possibilidade e de maneira a reduzir a utilização de recursos os serviços de balanceamento podem ser instalados no próprio *cluster* Kubernetes. Como foi sugerido no *cluster* mínimo de produção no diagrama da Figura 4.6 o balanceamento do tráfego é efetuado pelo balanceador HAproxy [83]. Este é instalado nos três nós do *cluster* e utiliza um ficheiro de configuração que se encontra no Anexo D onde são configurados os parâmetros e endereços para proceder ao balanceamento e redirecionamento dos pedidos pelos vários nós do *cluster*.

Para que no caso de falha do nó que está a assumir as funções de balanceamento outro nó possa assumir essas funções é utilizado o *software* Keepalived [132]. De maneira a configurar o Keepalived é editado o ficheiro `/etc/keepalived/keepalived.conf` no Node 1 que é configurado como primário devido ao campo *priority* ser maior que os restantes nós do *cluster*, como se pode ver a seguir:

```
vrrp_script chk_haproxy {
        script "killall -0 haproxy"
        interval 2
        weight 2
}

vrrp_instance VI_1 {
        interface eth0
        state MASTER
        virtual_router_id 51
        priority 101
        virtual_ipaddress {
            <VIP-ADDRESS>
        }
        track_script {
```

```
            chk_haproxy
        }
}
```

Nos restantes nós são colocadas as seguintes configurações:

```
vrrp_script chk_haproxy {
        script "killall -0 haproxy"
        interval 2
        weight 2
}

vrrp_instance VI_1 {
        interface eth0
        state MASTER
        virtual_router_id 51
        priority 100  # 100 no Node 2, 99 no Node 3
        virtual_ipaddress {
            <VIP-ADDRESS>
        }
        track_script {
            chk_haproxy
        }
}
```

Tanto no caso do nó primário como no caso dos nós secundários deve ser configurado um único endereço IP virtual (campo `<VIP-ADDRESS>`) que é atribuído consoante o nó que está ativo no momento.

4.6.3 Elasticidade nos nós do *Cluster* Kubernetes

O ambiente de produção tem um comportamento inconstante ao nível da procura de recursos o que faz com que a infraestrutura subjacente tenha de ser elástica para se adaptar às exigências dessa procura. No OpenStack para obtermos uma estrutura elástica composta por máquinas virtuais temos de recorrer ao módulo Heat, responsável pela orquestração da infraestrutura. Como foi visto na Secção 3.1.4.1, este módulo permite a partir de *templates*, previamente definidos, lançar e manipular elementos e recursos virtuais sobre a infraestrutura OpenStack. Um destes recursos virtuais é o `OS::Heat::AutoScalingGroup` [133]. Este recurso permite criar grupos de máquinas virtuais e associar a estes alarmes e elementos de controlo que de forma automática aumentam ou diminuem o número de máquinas virtuais pertencentes ao grupo. As decisões, levadas a cabo pelos elementos de controlo, tem por base informação métrica obtida a partir do módulo Ceilometer do OpenStack. Para fornecer elasticidade ao *cluster* Kubernetes vamos utilizar esta estratégia.

Para fornecer elasticidade ao ambiente Kubernetes vamos associar ao *cluster* mínimo de produção (falado na secção 4.6.2) um grupo de auto escalabilidade que permite que novos *nodes* sejam associados ao *cluster*. O diagrama da Figura 4.8 representa esse cenário.

Quando o *cluster* Kubernetes é criado sobre um ambiente de *cloud*, que é o caso desta dissertação, cada nó do *cluster* é uma máquina virtual. Então de maneira a acrescentar valor a esta implementação de escalabilidade não nos vamos apenas focar em acrescentar escalabilidade ao *cluster* Kubernetes mas sim a qualquer *cluster* que seja composto por máquinas virtuais. Pois o esforço dedicado a esta tarefa de implementação torna-se mais significativo quando é criado para qualquer ambiente composto por máquinas virtuais. Esta opção de escalabilidade foi implementada na CLI de controlo da *cloud* interna da AlticeLabs chamada pcloud abordada na Secção 3.2.3.

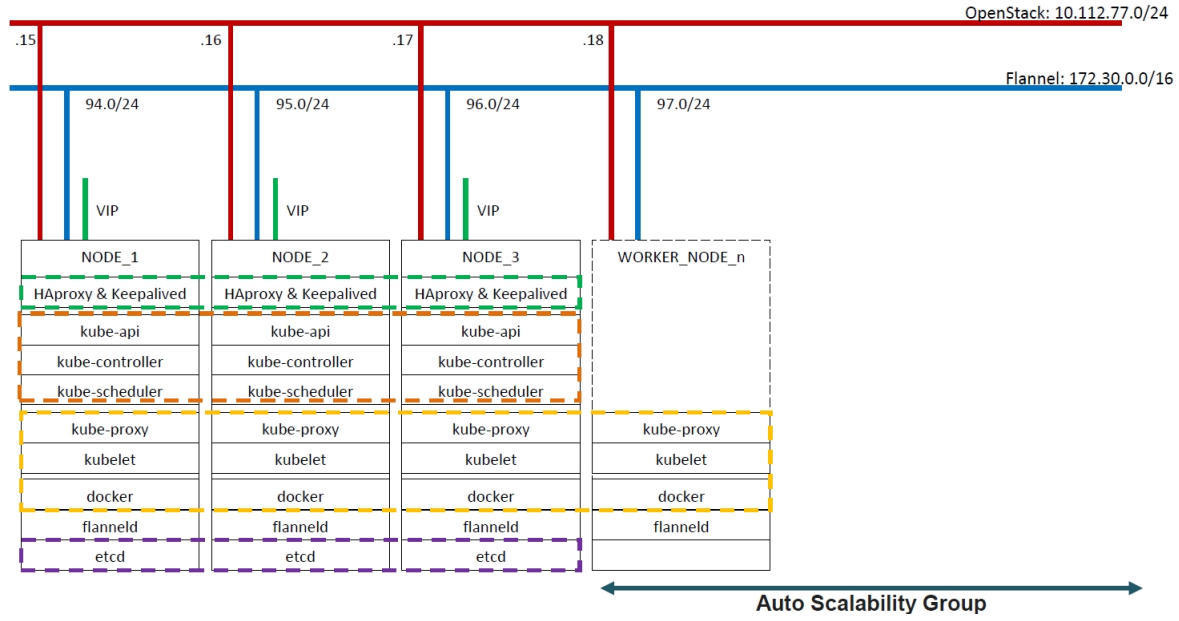


Figura 4.8: Cluster Kubernetes com a introdução do grupo de escalabilidade automática composto por *Worker Nodes*.

Como referido na Secção 3.2.3, a *framework* pcloud foi criada com o objetivo de simplificar o controlo e gestão da plataforma de *cloud* OpenStack da AlticeLabs, reunindo as funcionalidades que são necessárias ao nível da infraestrutura num ambiente de desenvolvimento e produção de *software*. Mas esta *framework* sendo um "*work in progress*" ainda não tinha a funcionalidade de criar grupos elásticos de máquinas virtuais de forma simples. Então para aumentar as capacidades desta *framework* e para auxiliar os trabalhos que se seguem nesta dissertação ao nível do Kubernetes em alta disponibilidade (como referido na secção 1.2) era necessário que o pcloud conseguisse, de forma simples, criar grupos de máquinas virtuais que tendo em conta a carga de *cpu* dessas máquinas virtuais pudesse ajustar automaticamente os tamanhos dos grupos.

Nesta secção será abordado todo o processo por de trás da funcionalidade acrescentada ao pcloud e ao cluster Kubernetes, implementada no âmbito desta dissertação. Vamos começar por descrever a interface e os comandos que permitem controlar os grupos de auto-escalabilidade. Logo de seguida vamos abordar a estratégia desenvolvida para abstrair todo o processo de criação de grupos de auto-escalabilidade utilizando os elementos e recursos disponíveis do OpenStack. No final desta secção será demonstrado o funcionamento das novas funcionalidades acrescentadas

ao pcloud com a criação de um grupo de auto escalabilidade que será associado ao cluster Kubernetes.

4.6.3.1 Proposta de Interface

De maneira a referir e invocar os grupos de auto-escalabilidade dentro da estrutura de comandos do pcloud tinha de se criar um novo comando com um nome simples que identificasse facilmente esta nova funcionalidade. O nome criado foi `asgroup` que surge como uma sigla para a expressão *Auto Scalability Group*. Ao nível das ações a realizar sobre o objeto `asgroup` o utilizador tem de ser capaz de realizar as seguintes ações: criar, listar e eliminar. Estas são as três ações básicas de qualquer recurso, pois permite que o utilizador o manipule durante todo o seu ciclo de vida. Ou seja, o utilizador é capaz de criar grupos de auto escalabilidade, logo de seguida pode listar e obter informações à cerca de todos os grupos que criou e por fim pode eliminá-los. Tendo em conta estes requisitos foram criados os seguintes comandos:

- **Criar:** `pcloud asgroup launch`
- **Listar:** `pcloud asgroup list`
- **Eliminar:** `pcloud asgroup delete`

De seguida será descrito a forma de invocação de cada comando assim como as suas *flags* e *outputs*.

4.6.3.2 Criar Grupos de Auto-Escalabilidade

O comando "asgroup launch" permite criar um grupo composto por um determinado número de máquinas virtuais que automaticamente, tendo em conta certas regras, aumenta ou diminui o número de instâncias melhorando a performance global das aplicações.

```
$ pcloud asgroup launch <group-name> [flags]
                                     [--min-size=<value>]
                                     [--max-size=<value>]
                                     [--scale-interval=<value>]
                                     [--evaluation-period=<value>]
                                     [--threshold-high=<value>]
                                     [--threshold-low=<value>]
                                     [--flavor=<value>]
                                     [--image=<value>]
                                     [--script-file=<file>]
```

Opções

- [–**min-size=<value>**]**:** Tamanho mínimo do grupo. O valor *default* é 1.
- [–**max-size=<value>**]**:** Tamanho máximo do grupo. O valor *default* é 5.

- [–**scale-interval=<value>**]: A quantidade de tempo, em segundos, entre atividades de *scaling*. O valor *default* é 120 segundos e o valor mínimo é 60 segundos.
- [–**evaluation-period=<value>**]: A quantidade de tempo, em segundos, que um recurso é avaliado considerando um dado *threshold*. O valor *default* é 120 segundos e o valor mínimo é 60 segundos.
- [–**threshold-high=<value>**]: O valor máximo médio de *threshold* para o recurso avaliado. Permite definir a percentagem de CPU máxima a partir da qual serão lançadas novas instâncias ao grupo. O valor *default* é 85
- [–**threshold-low=<value>**]: O valor mínimo médio de *threshold* para o recurso avaliado. Permite definir a percentagem de CPU mínima a partir da qual serão eliminadas instâncias do grupo. O valor *default* é 45
- [–**flavor=<value>**]: Tamanho/características das instâncias que vão pertencer ao grupo. O *flavor default* é aquele que é definido no ficheiro de configuração `pcloud.conf`.
- [–**image=<value>**]: Sistema Operativo a utilizar como base para as instâncias que vão pertencer ao grupo. A *image default* é a definida no ficheiro de configuração `pcloud.conf`.
- [–**script-file=<file>**]: Ficheiro *script* com instruções a serem executadas no *boot* das instâncias.

Output

O output do comando "pcloud asgroup launch" mostra informação relativa ao estado final da ação, ou seja, se foi concretizada com sucesso (CREATE_COMPLETE) ou insucesso (CREATE_FAIL). E para além disso lista os endereços IP's de cada máquina virtual criada no grupo.

4.6.3.3 Eliminar Grupos de Auto-Escalabilidade

O sub-comando "asgroup delete" permite eliminar um grupo de auto escalabilidade criado anteriormente. Esta operação destrói de forma irreversível toda a informação e elementos do grupo.

```
$ pcloud asgroup delete <group-name> --force
```

A *flag* –**force** serve de salvaguarda para o utilizador ter consciência que esta operação destrói de forma irreversível toda a informação e dados do grupo.

Output

Na mesma lógica que o comando anterior no final da ação de `delete` é mostrada uma mensagem de sucesso (DELETE_COMPLETE) ou então de insucesso (DELETE_FAIL).

4.6.3.4 Listar Grupos de Auto-Escalabilidade

O subcomando "asgroup list" permite obter a lista de grupos criados anteriormente e alguma informação relativa a estes.

```
$ pcloud asgroup list [asgroup-name] [--long]
```

Opções

- **[asgroup-name]:** Permite filtrar a informação por um `asgroup-name` específico.
- **[-l, –long]:** Mostra informação adicional relativa aos `asgroups` criados.

Output

A opção –**long** permite obter informação extra relativa a cada grupo. Quando a *flag* –long não está presente apenas é listado o nome e o *status* de todos os grupos existentes. Quando a *flag* –long está presente é mostrada informação extra relativa aos grupos existentes, tais como tamanho corrente, parâmetros e o endereço IP de cada uma das instâncias que pertencem ao grupo. Os parâmetros listados são todos os parâmetros (*flags*) que são comuns a todas as máquinas do grupo, que são: min-size, max-size, threshold-low, threshold-high, evaluation-period, scale-interval, flavor e image.

A razão pela qual existem um modo de listagem simples e outro complexo (com `--long`) é devido ao número de chamadas à API do OpenStack que se tem de realizar para obter certos níveis de informação mais detalhada. Como veremos mais à frente neste texto (na secção 4.6.3.5), a informação do nome e do *status* dos asgroups é possível obter apenas numa chamada. Mas a restante informação apenas é possível obter iterativamente (grupo a grupo) o que leva mais tempo. Então, devido a isto decidiu-se ter dois modos de listagem um rápido e simples e outro mais lento com mais informação.

Por uma questão de eficiência e de rapidez também é possível filtrar a lista ao fornecer o nome do grupo que queremos listar. Neste caso apenas é listada informação relativa a esse grupo.

4.6.3.5 Desenvolvimento dos Novos Comandos

Para desenvolver os comandos que criam a interface que manipulam os objetos `asgroup` foram adicionados os seguintes ficheiros ao *source code* do pcloud:

- `asgroup.go`: Este ficheiro vai permitir organizar o novo subcomando `asgroup` na árvore de comandos existente do pcloud através da biblioteca Cobra (abordada na Secção 3.2.3). Para além disso, vai permitir adicionar os subcomandos `launch`, `delete` e `list` ao comando `asgroup`.

- `asgroup_launch.go`: Este ficheiro vai implementar a funcionalidade do subcomando `launch`.
- `asgroup_delete.go`: Este ficheiro vai implementar a funcionalidade do subcomando `delete`.
- `asgroup_list.go`: Este ficheiro vai implementar a funcionalidade do subcomando `list`.

pcloud asgroup launch

Para a implementação do subcomando `launch` começamos por criar as *flags* através da biblioteca Cobra (abordada na secção 3.2.3). Por exemplo para criar e associar a flag `-script-file` ao comando `asgroup` é utilizado o seguinte excerto de código:

```
asgLaunchCmd.Flags().StringVarP(&scriptFileFlagValue, scriptFileFlag.Name,
    scriptFileFlag.Short, "", "Script file to run at instance boot time.")
```

Depois de todas as *flags* terem sido criadas, tanto os argumentos como as *flags* passam por um conjunto de validações de maneira a que o utilizador não introduza valores anómalos. Uma das verificações efetuadas é verificar por exemplo se já existe um grupo de auto escalabilidade criado com o mesmo nome que o introduzido.

Como referido anteriormente na introdução desta secção os grupos de auto escalabilidade vão ser criados a partir do módulo Heat do Openstack que é responsável pela orquestração da infraestrutura. Esta orquestração é realizada tendo em conta regras e declarações definidas num *template* (ver Anexo A) que é *hard coded* no código criado.

Uma das partes mais importantes do *template* é o excerto mostrado imediatamente a seguir.

```
 cluster:
   type: OS::Heat::AutoScalingGroup
   properties:
     resource:
       type: OS::Nova::Server
       properties:
         image: { get_param: image }
         flavor: { get_param: flavor }
         private_network: {get_param: private_network}
         public_network: {get_param: public_network}
         metadata: {"metering.stack": {get_param: "OS::stack_id"},
             "ASG_SERVER": "true"}
         user_data: |
           {{.ScriptFile}}
     min_size: { get_param: min_size }
     max_size: { get_param: max_size }
```

Como podemos ver, este excerto é responsável por criar um grupo de auto escalabilidade

representado pelo `type: OS::Heat::AutoScalingGroup` [133] ao qual os recursos que vão ser associados a esse grupo serão `OS::Nova::Server` [134] ou seja máquinas virtuais. Cada um destes *Servers* terá um conjunto de propriedades que são definidas no campo `properties`. Estas propriedades são definidas por campos pares chave-valor ao qual o valor de cada um destes campos será mapeado através do método `{get_param: VALUE}`. Esta função de mapeamento é executada através do próprio OpenStack tendo em conta os atributos definidos logo no início do template mostrado no Anexo A.

De forma a incluir o ficheiro *script* da *flag* `-file-script` é utilizado o campo `user_data`. O campo `user_data` é o campo que é responsável por incluir um *script* executável no *boot* de uma instância [134]. A linguagem Go tem a noção de templates e devido a isso mapeia estes para uma estrutura de dados que permite realizar alterações a partir de anotações no momento de execução através do *Parser* [135, 136]. É através da anotação `{{.ScriptFile}}` que o *script* do utilizador é inserido no *template* a enviar para o Openstack.

De maneira a controlar as dimensões do grupo são utilizados os campos `min_size` e `max_size` cujos os seus valores são definidos nos atributos iniciais do *template*. Para que o grupo de escalabilidade possa variar em tamanho dentro dos limites máximos e mínimos é necessário associar alarmes. Estes alarmes são definidos pelo seguinte excerto do *template*:

```
CPUAlarmHigh:
  type: OS::Ceilometer::Alarm
  properties:
    meter_name: cpu_util
    statistic: avg
    period: { get_param: evaluation_period }
    evaluation_periods: '1'
    threshold: { get_param: threshold_high }
    alarm_actions: [{get_attr: [scale_up_policy, alarm_url]}]
    matching_metadata: {'metadata.user_metadata.stack': {get_param:
        "OS::stack_id"}}
    comparison_operator: gt

CPUAlarmLow:
  type: OS::Ceilometer::Alarm
  properties:
    meter_name: cpu_util
    statistic: avg
    period: { get_param: evaluation_period }
    evaluation_periods: '1'
    threshold: { get_param: threshold_low }
    alarm_actions: [{get_attr: [scale_dw_policy, alarm_url]}]
    matching_metadata: {'metadata.user_metadata.stack': {get_param:
        "OS::stack_id"}}
    comparison_operator: lt
```

Os alarmes que são associados aos grupos de escalabilidade pertencem ao módulo Ceilometer

do OpenStack. Como foi falado na Secção 3.1.4.2, este é o módulo responsável pela telemetria do *cluster* OpenStack, que retira amostras que permitem entre outras coisas despoletar alarmes. Então os alarmes criados são do tipo `OS::Ceilometer::Alarm` que vão recolher a métrica `cpu_util` que é carga de CPU de cada uma das instâncias do grupo. Por *default* esta métrica é recolhida a cada 600 segundos mas para tornar o sistema mais sensível foi alterada a configuração para recolher métricas a cada 60 segundos. Esta alteração foi efetuada no ficheiro `/etc/ceilometer/ceilometer.conf` no campo `evaluation_interval = 60` e no ficheiro `/etc/ceilometer/ceilometer.conf` nos campos `interval` [137].

No `CPUAlarmHigh` será calculada a média (`statistic: avg`) das métricas de CPU do grupo de instâncias e caso este valor seja maior (`comparison_operator: gt`) que o valor do `threshold` durante um determinado período de tempo (`period: get_param: evaluation_period`) será realizada a ação definida em `alarm_actions: [get_attr: [scale_up_policy, alarm_url]]`. No caso do `CPUAlarmLow` é semelhante ao `CPUAlarmHigh`, a diferença é que neste caso estamos avaliar se a carga de CPU é menor (`comparison_operator: lt`) do que o `threshold` e a ação despoletada pelo alarme vai ser efetuada pela política `scale_dw_policy`.

As políticas anteriormente faladas são definidas à custa do seguinte excerto do *template*:

```
scale_up_policy:
  type: OS::Heat::ScalingPolicy
  properties:
    adjustment_type: change_in_capacity
    auto_scaling_group_id: {get_resource: cluster}
    cooldown: { get_param: scale_interval }
    scaling_adjustment: '1'

scale_dw_policy:
  type: OS::Heat::ScalingPolicy
  properties:
    adjustment_type: change_in_capacity
    auto_scaling_group_id: {get_resource: cluster}
    cooldown: { get_param: scale_interval }
    scaling_adjustment: '-1'
```

Em cada um dos casos a ação será a alteração da capacidade do grupo (`adjustment_type: change_in_capacity`) incrementando ou decrementando apenas uma unidade (`scaling_adjustment`) o *cluster* de máquinas virtuais. Entre cada ação despoletada existe um intervalo de tempo definido no campo `cooldown`.

Por fim após o *template* estar criado e todos os parâmetros estarem devidamente validados é criado o grupo de escalabilidade automática à custa do seguinte excerto de código:

```
  asgroup := stacks.CreateOpts{
    Name:       asgName,
    TemplateOpts: template,
```

```
        Parameters: param,
        DisableRollback: stacks.Enable, //In case of stack create fail: delete
            stack
    }
    //Create Auto Scaling Group
    asgroup, err := stacks.Create(orchestrationClient, asgroup).Extract()
```

O pedido de criação do grupo de escalabilidade automática é feito à custa da biblioteca GopherCloud (abordada na Secção 3.2.3) pela função `Create` que interage diretamente com a API do OpenStack.

pcloud asgroup delete

Para a criação do subcomando `delete` e com semelhança ao comando `launch`, abordado anteriormente, este é inserido na árvore de comandos do pcloud através da biblioteca Cobra (abordada na secção 3.2.3). Com esta biblioteca definimos a função que implementa o comando (`deleteASGroup`) e para além disso permite incluir as descrições textuais da utilização do comando nos formatos *Short* e *Long*, como se pode ver pelo o excerto de código incluído a seguir:

```
  func NewASGroupDeleteCommand() *cobra.Command {
   asgDeleteCmd := &cobra.Command{
      Use:   "delete <asgroup-name>",
      Short: "Delete a entire Auto Scaling Group.",
      Long: "Permanently delete a Auto Scaling Group. This operation isn't
          reversible, use with care.",
      PreRun: validateDelASGroupArgs,
      Run: deleteASGroup,
    }
    asgDeleteCmd.Flags().BoolVarP(&forceFlagValue, forceFlag.Name,
        forceFlag.Short, false, "Force delete")

    return asgDeleteCmd
}
```

Para validar se o nome do `asgroup` inserido é válido e existe são efetuadas validações à custa da função `validateDelASGroupArgs`, definida no `PreRun` da chamada à biblioteca Cobra.

Como a ação de eliminar um grupo de escalabilidade automática deve ser feita com uma certa cautela porque elimina permanentemente todos os recursos e dados é pedido ao utilizador que insira a flag `-force`, pois caso contrário aparece a seguinte mensagem:

`"This action will destroy the entire Auto Scaling Group and isn't reversible. Specify -force if sure!"`

Por fim, após todos as validações terem sido realizadas e depois de ser obtido o identificador do asgroup (obtido através do nome do asgroup) é dado a ordem de `delete` através da função

"Delete" do GopherCloud. Isso é feito através do seguinte excerto de código:

```
stacks.Delete(orchestrationClient, asgName, asgroup.ID)
```

pcloud asgroup list

A listagem dos `asgroups` é apresentada numa tabela em dois modos diferentes, um modo simples e um detalhado. Como é possível ver pelo excerto de código a seguir, o modo simples apenas lista o nome e o *status* de cada grupo de escalabilidade automática. Enquanto que o modo detalhado, que é dado pela *flag* `-long` ou `-l`, apresenta os seguinte campos: nome, status, tamanho corrente, parâmetros de criação do grupo e os endereços IP's associados a cada instância.

```
    table := tablewriter.NewWriter(os.Stdout)

   if (longListFlagValue) {
      table.SetHeader([]string{"name", "status","current-size", "parameters",
          "instances"})
   } else {
      table.SetHeader([]string{"name", "status"})
   }
```

Foi decidido criar dois modos de listagem por uma questão de eficiência. Pois para o modo de listagem simples com apenas uma chamada à API são retornados o nome e o status de todos os `asgroups`, como se pode ver pelo excerto imediatamente a seguir:

```
...
    //List All Stacks
   asgPager := stacks.List(orchestrationClient, listOpts)
   singlePage, err := asgPager.AllPages()
   matchingASGroup, err := stacks.ExtractStacks(singlePage)
...
    for i := range matchingASGroup {
      asg := matchingASGroup[i]
      ...
      row := []string{asg.Name, asg.Status}
       table.Append(row)
   }
...
```

Para o modo de listagem detalhado é necessário que em ciclo e para cada `asgroup` recebido na primeira chamada à API (mostrada em cima) fazer um novo pedido, como se pode ver pelo excerto a seguir:

```
...
```

```
  for i := range matchingASGroup {
    ...
    if longListFlagValue {
      //Get and extract info about specific stack
     st := stacks.Get(orchestrationClient,asg.Name,asg.ID)
     asgDetail, err := st.Extract()
     ...
    }
  ...
  }
...
```

Por cada um destes pedidos específicos para cada `asgroup` é devolvida a seguinte informação: nome, *status*, tamanho corrente, *min_size*, *max_size*, *threshold_low*, *threshold_high*, *evaluation_period*, *scale_interval*, *flavor*, *image* e endereços IP's. Esta informação é organizada e apresentada nas colunas adicionais.

4.6.3.6 Demonstração de Utilização - pcloud e Kubernetes

Nesta secção vamos demonstrar o funcionamento dos novos sub-comandos acrescentados ao pcloud no âmbito desta dissertação. Como foi dito anteriormente, estas funcionalidades foram acrescentadas à *framework* pcloud com o objetivo de fornecer essencialmente elasticidade ao *cluster* Kubernetes composto por máquinas virtuais.

Então para acrescentar os mecanismos de elasticidade ao *cluster* Kubernetes foi dado o seguinte comando pcloud:

```
$ pcloud asgroup launch ctidois-k8s --image=ctidois-init-snap --flavor=fl.large
    --min-size=1 --max-size=5 --scale-interval=60 --evaluation-period=60
    --threshold-low=40 --threshold-high=80 --script-file=./script.sh
```

Com o comando anterior pretende-se lançar um grupo de escalabilidade automática (`asgroup`) chamado ctidois-k8s com uma imagem CentOS 7 com o cloud-init instalado chamada `ctidois-init-snap` e com um flavor `fl.large` com as características de 4 GB de RAM, 30 GB de disco e 2 vCPUs. O grupo será criado com uma instância inicial e pode crescer até um tamanho máximo de cinco instâncias. As atividades de escalabilidade serão efetuadas em intervalos de 60 segundos quando durante 60 segundos a carga de CPU média das máquinas que compõe o grupo for superior a 80% ou inferior a 40%.

No output do comando é listado o endereço IP da instância criada, como se pode ver a seguir:

```
admin_internal_net:192.168.14.233; floating-ip:10.112.76.186
```

Para que as máquinas virtuais que pertencem a este grupo de escalabilidade possam se ligar ao *cluster* Kubernetes e consequentemente sejam configuradas num *Worker Node* é fornecido o

ficheiro `script.sh`, que pode ser visto no Anexo C. É a partir deste *script* que os pacotes do Docker e Kubernetes são instalados e configurados assim como os endereços de acesso e chaves são fornecidos ao novo *node*. Para que o *script* seja executado no tempo de *boot* de cada instância é necessário que a imagem base com o sistema operativo tenha o pacote `cloud-init` instalado [138].

Após a máquina ter sido inicializada e os pacotes terem sido totalmente instalados e configurados esta passa a pertencer ao *cluster* Kubernetes. A melhor maneira de verificar isso é listar a lista de nós que compõe o *cluster* Kubernetes através do seguinte comando mostrado na Figura 4.9.

```
[dreis@localhost ~]$ kubectl get nodes -o wide
NAME                                                STATUS   AGE   EXTERNAL-IP
ct-4x5u-xyozcpxhwztk-d5c4mtyef357-server-l3c4262zaifw   Ready    46s   10.112.76.186
ctidois-kmaster1                                    Ready    43d   10.112.76.180
ctidois-kmaster2                                    Ready    9d    10.112.76.181
ctidois-kmaster3                                    Ready    24d   10.112.76.183
```

Figura 4.9: Listagem dos nós do *cluster* Kubernetes contendo o recente *Worker Node* pertencente a um grupo de escalabilidade automática.

Como se pode ver pela figura 4.9 foi adicionado um novo *Worker Node* com o nome `ct-4x5u-xyozcpxhwztk-d5c4mtyef357-server-l3c4262zaifw`.

Para despoletar uma atividade de *scaling* automático foi dado o comando

```
$ while[1]; do echo $((13**99)) 1>/dev/null 2>&1; done &
```

com o objetivo de elevar a carga de CPU para próximo dos 100% num contentor que estava a executar no novo *Worker Node*. Após 60 segundos de análise o OpenStack lançou automaticamente outro Worker Node novo. Com esta ação o grupo de escalabilidade automática passou a ter uma dimensão de 2 elementos, como se pode ver pela execução do comando `list` na Figura 4.10. Com o fim de tempo de *boot*, o novo *Worker Node* é registado junto do *cluster* e este passa a parecer na lista de elementos obtida pelo comando `kubectl`, como se pode ver pela Figura 4.11.

```
[dreis@localhost ~]$ pcloud asgroup list -l
+-------------+-----------------+--------------+--------------------------+-----------------------------------------------------------+
|    NAME     |     STATUS      | CURRENT-SIZE |        PARAMETERS        |                         INSTANCES                         |
+-------------+-----------------+--------------+--------------------------+-----------------------------------------------------------+
| ctidois-k8s | CREATE_COMPLETE |            2 | min-size: 1              | admin_internal_net:192.168.14.234;floating:10.112.76.162  |
|             |                 |              | max-size: 5              | admin_internal_net:192.168.14.233;floating:10.112.76.186  |
|             |                 |              | threshold-low: 25        |                                                           |
|             |                 |              | threshold-high: 80       |                                                           |
|             |                 |              | evaluation-period: 60    |                                                           |
|             |                 |              | scale-interval: 60       |                                                           |
|             |                 |              | flavor: fl.large         |                                                           |
|             |                 |              | image: ctidois-init-snap |                                                           |
|             |                 |              |                          |                                                           |
+-------------+-----------------+--------------+--------------------------+-----------------------------------------------------------+
```

Figura 4.10: Execução do comando pcloud asgroup list onde é possível ver que houve uma ação de escalabilidade no *cluster* e este passou a ter dimensão 2.

```
[dreis@localhost ~]$ kubectl get nodes -o wide
NAME                                                STATUS   AGE   EXTERNAL-IP
ct-4x5u-egwcvgz2byou-bz6nmb4kegpy-server-krnao6fyiv7q   Ready    5s    10.112.76.162
ct-4x5u-xyozcpxhwztk-d5c4mtyef357-server-l3c4262zaifw   Ready    6m    10.112.76.186
ctidois-kmaster1                                    Ready    43d   10.112.76.180
ctidois-kmaster2                                    Ready    9d    10.112.76.181
ctidois-kmaster3                                    Ready    24d   10.112.76.183
```

Figura 4.11: Listagem dos *nodes* do *cluster* Kubernetes após uma ação de escalabilidade onde é possível ver que existem 2 *Worker Nodes*.

Capítulo 5

Estudo de Caso

Neste Capítulo vamos avaliar as abordagens anteriormente sugeridas para a gestão e *deployment* de aplicações conteinerizadas. Para proceder às avaliações foi utilizada uma aplicação numa arquitetura baseada em micro-serviços selecionada tendo em conta um conjunto de requisitos. Inicialmente vamos contextualizar o leitor sobre a aplicação que vamos conteinerizar e o porquê dessa aplicação ter sido selecionada para efetuar o estudo de caso. Após essa contextualização serão enumeradas as alterações que foram necessárias realizar nessa aplicação de maneira a que esta possa tirar partido de todas as utilidades de um ambiente Kubernetes. Com as alterações concluídas será descrito o modo como as imagens Docker foram criadas para cada serviço que compõe a aplicação selecionada. Depois das imagens terem sido criadas e disponibilizadas no repositório poderá ser iniciado o *deployment* da aplicação no cluster Kubernetes esquematizado anteriormente na Secção 4.6.2. Nesta parte do texto serão descritas as configurações e os ficheiros criados para realizar o *deployment* da aplicação para se dar inicio à realização dos testes e avaliações. Nesta fase vamos realizar um conjunto de testes que vão permitir avaliar a plataforma Kubernetes em vários aspetos. A partir destes testes vão ser obtidos um conjunto de resultados que serão discutidos no próximo Capítulo.

5.1 Pontos de Avaliação e Requisitos

A aplicação que deve ser selecionada tem de ser baseada numa arquitetura em micro-serviços (como falado na Secção 2.3) e conter um conjunto de requisitos mínimos para podermos avaliar as propostas apresentadas no Capítulo anterior para o *deployment* e gestão de aplicações conteinerizadas. Os pontos que devem ser avaliados são:

1. ***Deployment*:** avaliar se o orquestrador Kubernetes é capaz de realizar o *deployment* total dos vários contentores que compõe a aplicação;
2. **Descoberta de Serviços:** avaliar se o sistema de descoberta de serviço do Kubernetes é capaz de fornecer essa descoberta aos serviços da aplicação;
3. **Persistência de Dados:** avaliar se a utilização de Persistent Volumes (abordados na

Secção 3.4.3.2) conseguem fornecer persistência aos dados mesmo quando um contentor morre e é relançado noutro nó do *cluster*;

4. **Recuperação de Falhas:** avaliar os mecanismos de recuperação de falhas do Kubernetes. Isto é, quando os serviços da aplicação morrem ou caso um dos nós do *cluster* falhe perceber se o Kubernetes é capaz de relançar os contentores afetados e até que ponto o serviço da aplicação é afetado;
5. **Configurações da Aplicação:** avaliar se o Kubernetes é capaz de injetar as configurações necessárias de uma forma conveniente para o arranque da aplicação;
6. **Elasticidade:** avaliar se na necessidade de mais recursos se o grupo de escalabilidade automática lançado pelo pcloud é capaz de relançar mais nós ao *cluster* Kubernetes para albergar novos contentores;
7. **Variação do Número de Réplicas:** avaliar os impactos provocados pela variação do número de réplicas na resposta dada aos pedidos dos clientes;
8. **Atualizações:** avaliar os impactos provocados pelas atualizações ao *software* na resposta aos pedidos dos clientes;

Para poder auferir os pontos de avaliação anteriormente apresentados a aplicação deve ter os seguintes requisitos:

(A) ter uma arquitetura modular composta por vários serviços. Com este requisito pretende-se avaliar os pontos 1, 2 e 6;
(B) as configurações de arranque da aplicação são obtidas de um ponto central. Com este requisito pretende-se avaliar os pontos 2 e 5;
(C) ter serviços *stateless* com a possibilidade de ter várias réplicas do mesmo serviço em funcionamento. Com este requisito pretende-se avaliar os pontos 4, 7 e 8;
(D) ter serviços *stateful* com necessidade de persistência de dados. Com este requisito pretende-se avaliar os pontos 3 e 4;

5.2 Aplicação SmartIoT

A aplicação selecionada que cumpre os requisitos e que vai permitir avaliar as abordagens anteriormente sugeridas para o *deployment* e gestão de aplicações conteinerizadas chama-se SmartIoT. Esta aplicação está a ser desenvolvida em Java com o objetivo de experimentar as tecnologias e os conceitos da *Internet of Things* (IoT). É uma plataforma de receção e distribuição de mensagens, idealizada de acordo com um conjunto de premissas ao nível do desempenho, fiabilidade e capacidade de integração, por forma a agregar valor num ambiente IoT de larga escala.

Esta aplicação foi criada no âmbito de outro projeto da AlticeLabs [4] que não o desta dissertação. Como tal, a escolha dos componentes assim como a organização dos mesmos não têm qualquer relação com esta dissertação. Apenas vamos conteinerizar e executar a aplicação num ambiente conteinerizado sem alterações ao nível da sua arquitetura de maneira a realizar os

testes enumerados anteriormente.

As principais funcionalidades da aplicação Smartiot são:

- Capacidade de acomodar diversos protocolos de integração com as entidades externas (eg., REST, CoAP, etc.);
- Receção de mensagens com garantias de durabilidade;
- Entrega de mensagens via *push* ou *polling*;
- Garantia de entrega de mensagens pela ordem de receção;

5.2.1 Exemplo de Utilização

De maneira a perceber melhor as possíveis utilidades que esta ferramenta permite é dado de seguida um exemplo. Como se pode ver pela Figura 5.1, este exemplo considera dois tipos de dispositivos, um ar condicionado e o seu controlo remoto. Neste cenário a comunicação e entrega de mensagens entre um aparelho de ar condicionado e o seu respetivo controlo remoto é mediada através da SmartIoT. O fluxo de funcionamento neste exemplo é:

O sistema de ar condicionado:

1. publica a temperatura corrente em intervalos regulares;
2. aceita comandos para aumentar/diminuir a temperatura ou ligar/desligar o sistema;
3. pode ser acedida através de uma interface HTTP;

O sistema de controlo remoto:

1. envia comandos de maneira a controlar o sistema de ar condicionado;
2. mostra a temperatura reportada pelo sistema de ar condicionado;

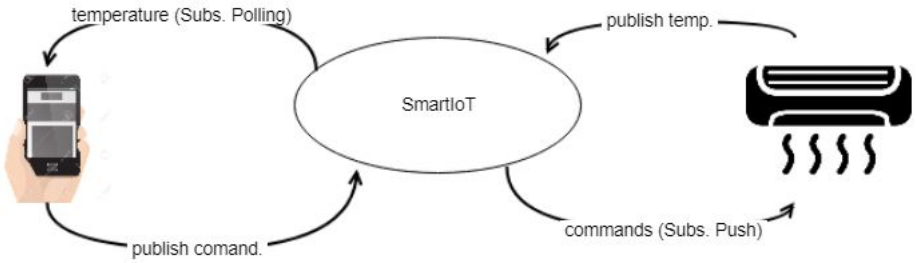


Figura 5.1: Exemplo de utilização básico da SmartIoT.

5.2.2 Arquitetura

A aplicação SmartIoT tem uma arquitetura modular com vários componentes onde cada um dos componentes tem uma funcionalidade bem definida, criando um serviço. Isto respeita o ponto (A) da lista de requisitos apresentada anteriormente. Estes componentes permitem à aplicação lidar com um grande volume de mensagens, entrega e armazenamento dessas mensagens. Na

Figura 5.2 pode ser vista a disposição de cada um deles.

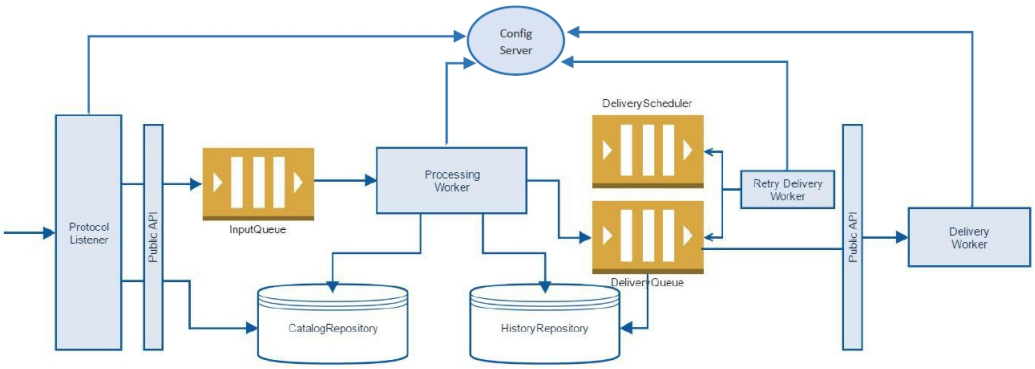


Figura 5.2: Arquitetura e componentes da SmartIOT.

De seguida vão ser explicados com algum detalhe cada um dos componentes que compõe a SmartIoT:

Config Server

O Config Server é o ponto central de configurações que é responsável por manter as configurações de todos os outros serviços da SmartIoT. O que faz com que os outros serviços tenham de aceder inicialmente ao Config Server de maneira a obter as configurações de arranque assim como os caminhos para os outros serviços. Este serviço está de acordo com o ponto (B) da lista de requisitos apresentada anteriormente.

Protocol Listener

O componente Protocol Listener consiste na camada protocolar de interação entre a plataforma e as entidades (aplicações, dispositivos, etc.). A plataforma suporta nativamente a capacidade de instanciação de diversos Protocol Listener o que vai de encontro ao ponto (C) da lista de requisitos apresentada anteriormente. O Protocol Listener utiliza a Public API para aceder às funcionalidades internas da plataforma.

Public API

Esta API disponibiliza um conjunto de operações de interação com os componentes nucleares da plataforma. Qualquer extensão à plataforma deverá ser efetuada tendo por base as operações disponibilizadas por esta API.

InputQueue

Componente responsável por armazenar as novas mensagens publicadas para a plataforma (*buffer* temporário), e que aguardam um processamento posterior pelos diversos componentes nucleares da plataforma. Este componente disponibiliza uma API através da qual é possível publicar e consumir mensagens.

Este componente é criado à custa do processador de *streams* de dados chamado Apache Kafka [139]. Esta plataforma recebe mensagens de produtores que podem ser a um ritmo elevado e

distribui estas mensagens por várias partições. Os consumidores ligam-se às partições de maneira a obter as mensagens que lhe são destinadas. Para além do Apache Kafka é utilizado o Apache Zookeeper [140] que fornece um ponto central de informação, configurações, sincronização e controlo ao *cluster* composto por nós de Kafkas.

Processing Worker

O processing worker é o componente interno responsável por consumir as mensagens existentes na Input Queue e efetuar um conjunto de atividades associadas: armazenar as mensagens em histórico, publicar as mensagens na Delivery Queue de acordo com as subscrições existentes. É possível escalar e instanciar diversos Processing Workers, tendo em conta os requisitos de desempenho a suportar. Esta funcionalidade respeita o ponto (C) da lista de requisitos apresentada anteriormente.

Delivery Queue

A Delivery Queue é o componente que armazena as mensagens pendentes para entrega a subscritores de mensagens. A entrega poderá ser efetuada por *push* de mensagens para os consumidores, ou através de operações de *polling* executadas pelos consumidores.

Este componente é criado com o Redis [141]. O Redis é um sistema *open-source* que permite o armazenamento em memória de pares chave-valor que pode ser utilizado como base de dados, *cache* e *broker* de mensagens com alta performance.

Delivery Scheduler

Sempre que ocorre um erro na tentativa de entrega de uma mensagem ao destinatário, é reagendada uma nova tentativa para uma data futura. Este componente tem como responsabilidade determinar qual o melhor instante em que uma mensagem deve ser reenviada.

Tal como a Delivery Queue a Delivery Scheduler é criada à custa do Redis [141].

Retry Delivery Worker

O Retry Delivery Worker é um componente interno responsável por consumir informação relativa a falhas de entrega existentes na Delivery Queue e reagendar novas tentativas no Delivery Scheduler de acordo com a política de reenvios definida na subscrição. Tem também a função de consumir os pedidos cujo instante de reenvio foi atingido e ressubmeter esses pedidos para a Delivery Queue novamente.

Delivery Worker

O componente Delivery Worker é responsável por aceder à Delivery Queue de maneira a obter e entregar as mensagens a um determinado consumidor. A entrega pode ser efetuada por *push* para os consumidores, ou então através de *polling* a partir dos consumidores.

Catalog Repository

A plataforma mantém um conjunto de informação relativo a entidades e demais artefactos do ecossistema (ex., aplicações, *devices*, *streams*, subscrições, etc.). Esses dados materializam o cadastro das entidades passíveis de integração com a plataforma. A gestão CRUD destes dados é efetuada pelo Catalog Repository.

O Catalog Repository é uma base de dados e devido a isso à criado através do sistema *open source* de gestão de bases de dados relacionais chamado PostgreSQL [142]. Este serviço está de acordo com o ponto (D) da lista de requisitos apresentada anteriormente.

History Repository

Todas as mensagens publicadas na plataforma são armazenadas num histórico pesquisável, correspondente ao componente History Repository.

Da mesma forma que o Catalog Repository o History Repository é uma base de dados criada através do PostgreSQL [142]. Este serviço também está de acordo com o ponto (D) da lista de requisitos apresentada anteriormente.

5.3 SmartIoT em Contentores

Depois de percebermos os componentes e o papel que cada um deles desempenha na aplicação vamos começar por expor um esquema que organiza cada um dos componentes numa arquitetura lógica suportada por contentores. Este é um dos passos que deve ser feito antes de passar à conteinerização propriamente dita pois vai permitir definir com alguma exatidão o número mínimo de contentores necessários de maneira a perceber a quantidade de imagens que vão ser criadas posteriormente.

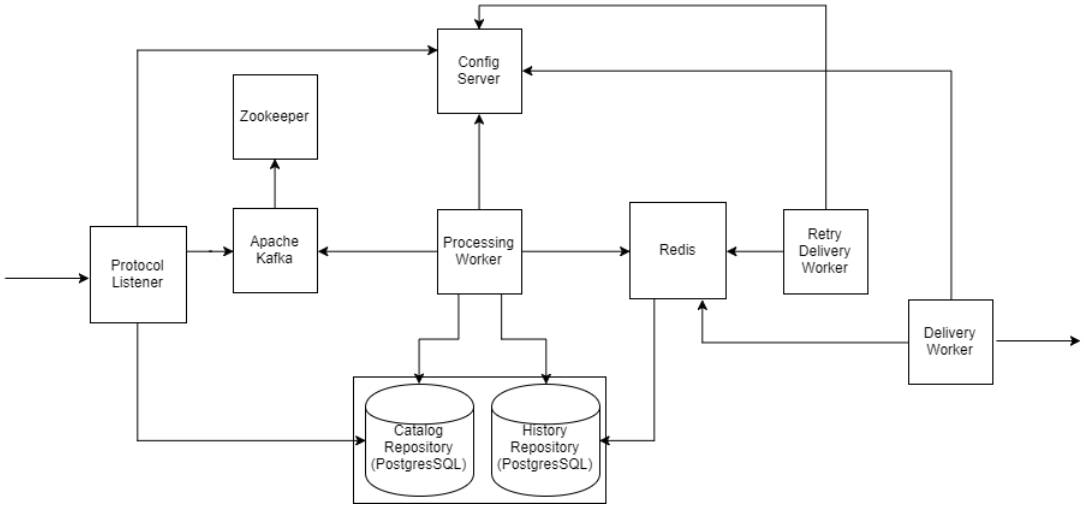


Figura 5.3: Distribuição dos componentes da SmartIoT por contentores.

Como podemos ver pela Figura 5.3 a SmartIoT para ser conteinerizada vai precisar de pelo menos 9 contentores distintos. Logo vai ser necessário criar 9 imagens para esses contentores.

Alterações Realizadas

Para que a aplicação SmartIoT pudesse ser conteinerizada era necessário realizar algumas alterações, de maneira a adaptar esta aplicação a um ambiente conteinerizado tirando partido de todas as funcionalidades do Kubernetes. Como esta aplicação originalmente foi criada numa arquitetura de micro-serviços e pensada para operar num ambiente distribuído as alterações necessárias foram realizadas num curto espaço de tempo com um esforço pouco significativo. Essas alterações foram:

- Originalmente na SmartIoT cada instância dos serviços antes do seu arranque teria de ser assinalada por um identificador. Para atribuir estes identificadores era necessário ter um número estático de instâncias para cada serviço. Num ambiente ágil e automaticamente escalável isto não poderia acontecer. Devido a isso este campo foi extraído das instâncias e estas passam a receber o mesmo identificador que é atribuído ao contentor pelo Kubernetes.
- Originalmente na SmartIoT os *logs* de cada um dos serviços estavam a ser escritos para ficheiros específicos que ficavam guardados na *host* em vez de serem enviados para o *standard output* ou *standard error*. Esta realidade não ia de encontro ao Kubernetes, pois neste orquestrador os *logs* que são apresentados são aqueles que são escritos para o *standard output* ou *standard error*. Devido a isto, foram feitas alterações para que todos os serviços do SmartIoT passassem a escrever os *logs* para o *standard output*.

Com estas alterações a SmartIoT estava pronta a ser conteinerizada e executada num ambiente Kubernetes.

5.4 Criação das Imagens

Após as alterações e a distribuição lógica dos serviços por contentores estar concluída, passamos à construção das imagens Docker. Como vimos anteriormente vai ser necessário construir pelo menos 9 imagens Docker, uma por cada serviço. Como ponto de partida para a construção das imagens vai ser necessário a criação da imagem base com o sistema operativo CentOS 6.

5.4.1 Imagem Base

A imagem que vai servir como base e ponto de partida para a imagem *framework* é constituída pelo sistema operativo CentOS 6. Para a construção desta imagem foi utilizado o processo falado no Capítulo anterior na Secção 4.3 e então foi utilizada a *script* [115] que é fornecida pelo projeto Moby [143] para auxiliar a criação da imagem. Então, para a construção da imagem basta executar esta *script* dentro de uma máquina com o CentOS 6 e com o Docker instalado. No final da execução do *script* vamos ter uma nova imagem com o CentOS 6 visível através do comando `docker images`. Após isso basta fazer *push* da imagem para o repositório de imagens e esta passa a estar disponível a todas as máquinas com acesso ao repositório.

5.4.2 Imagem Framework

Como foi sugerido na Secção 4.3 é necessário criar uma imagem *framework* onde vai estar todo o *software* que é comum à aplicação SmartIoT e que é partilhado por todos os serviços a que lhe pertencem.

O *software* que é comum a todos os serviços que compõe a SmartIoT é o Java e o JDK. Tendo em conta isso, foi criada a imagem base à custa da seguinte Dockerfile:

```
FROM repo-docker.example.com/centos6:6.8
RUN yum -y update; yum clean all
RUN yum -y install env-java1.8.noarch \
&& yum -y install jdk-ptin
CMD ["/bin/bash"]
```

A imagem *framework* tem como ponto de partida a imagem base com o sistema operativo CentOS 6 em que o seu processo de criação foi falado na secção anterior.

5.4.3 Imagens dos Serviços *Stateless*

Nesta secção é mostrada a criação das imagens de todos os serviços *stateless* que compõe a SmartIoT. É de notar que estas imagens têm todas como ponto de partida a imagem *framework* criada na secção imediatamente acima chamada `repo-docker.example.com/smartiot-framework`.

Config Server

A imagem para o serviço Config Server é criada à custa da seguinte Dockerfile:

```
FROM repo-docker.example.com/smartiot-framework
RUN yum -y install /data/smartiot-ff*
VOLUME ["/opt/ptin/smartiot/conf"]
EXPOSE 9090
ENTRYPOINT ["/opt/ptin/smartiot/bin/config-server.sh"]
CMD ["/opt/ptin/smartiot/conf/config-server.json"]
```

Este serviço é o ponto central de configurações que é exposto na porta 9090 para que outros contentores se possam ligar a este e obter as configurações de arranque. Para que o Config Server consiga fornecer as configurações este tem de ter acesso às mesmas. Para isso é criado um Volume de maneira a incluir esses ficheiros de configuração. O ponto de arranque desta imagem será o ficheiro `/opt/ptin/smartiot/bin/config-server.sh` e o argumento passado pela instrução `CMD` define qual o ficheiro de configuração que deve ser considerado, que neste caso é `/opt/ptin/smartiot/conf/config-server.json`.

No apêndice B existe um exemplo do ficheiro `config-server.json` que pode ser consultado de maneira a obter mais informações sobre as configurações disponíveis.

Protocol Listener

A imagem para o serviço Protocol Listener é criada à custa da seguinte Dockerfile:

```
FROM repo-docker.example.com/smartiot-framework
RUN yum -y install env-jruby2 && yum install -y jruby; yum clean all
RUN yum -y install /data/smartiot-http-listener* \
                   /data/smartiot-common* \
                   /data/smartiot-ff*
EXPOSE 3000
ENTRYPOINT ["/opt/ptin/smartiot/bin/http-listener.sh"]
CMD ["-c url_conf"]
```

Este serviço é exposto na porta 3000 do contentor e o *script* de arranque é o `/opt/ptin/smartiot/bin/http-listener.sh` com os argumentos `-c url_conf`. Ao qual o `url_conf` será o endereço de acesso ao Config Server de maneira a obter as configurações de arranque e que apenas será atribuído no momento do *deployment* do contentor.

Processing Worker

A imagem para o serviço Processing Worker é criada à custa da seguinte Dockerfile:

```
FROM repo-docker.example.com/smartiot-framework
RUN yum -y install /data/smartiot-processing-worker* \
                   /data/smartiot-common* \
                   /data/smartiot-ff*
ENTRYPOINT ["/opt/ptin/smartiot/bin/processing-worker.sh"]
CMD ["-c url_conf"]
```

O serviço Processing Worker não tem nenhuma porta exposta porque este contentor acede aos outros contentores e nenhum outro contentor a ele, como pode ser visto pela Figura 5.3. O ponto de arranque é `/opt/ptin/smartiot/bin/processing-worker.sh` com o argumento `-c url_conf` que permite definir o caminho até ao serviço Config Server de maneira a obter as configurações do ponto central.

Retry Delivery Worker

A imagem para o serviço Retry Delivery Worker é criada à custa da seguinte Dockerfile:

```
FROM repo-docker.example.com/smartiot-framework
RUN yum -y install /data/smartiot-retry-delivery-worker* \
                   /data/smartiot-common* \
                   /data/smartiot-ff*
ENTRYPOINT ["/opt/ptin/smartiot/bin/retry-delivery-worker.sh"]
CMD ["-c url_conf"]
```

O ponto de arranque é o ficheiro `/opt/ptin/smartiot/bin/retry-delivery-worker.sh` com os argumentos `-c url_conf` para se ligar ao Config Server de maneira a obter as configurações de arranque.

Delivery Worker

A imagem para o serviço Delivery Worker é criada à custa da seguinte Dockerfile:

```
FROM repo-docker.example.com/smartiot-framework
RUN yum -y install /data/smartiot-delivery-worker* \
                   /data/smartiot-common* \
                   /data/smartiot-ff*
ENTRYPOINT ["/opt/ptin/smartiot/bin/delivery-worker.sh"]
CMD ["-c url_conf"]
```

O arranque do contentor é feito à custa do ficheiro `/opt/ptin/smartiot/bin/delivery-worker.sh`. À semelhança dos serviços anteriores o argumento `-c url_conf` vai permitir definir o caminho até ao serviço Config Server para obter as configurações de arranque.

5.4.4 Imagens dos Serviços Stateful

Nesta secção vai ser abordado a forma como as imagens dos serviços *stateful* foram criados à custa das Dockerfiles. Como referido anteriormente na secção 5.2.2, os serviços *stateful* que fazem parte da SmartIoT são: Apache Kafka, Apache Zookeeper, Redis e PostgreSQL.

Apache Kafka

A imagem do Apache Kafka é criada à custa da seguinte Dockerfile:

```
FROM repo-docker.example.com/centos6:6.8
RUN yum -y install env-java1.8 && \
    yum -y install jdk-ptin
RUN yum -y install kafka-scala211-0.10.0.1
COPY start-kafka.sh /start-kafka.sh
RUN chmod +x /start-kafka.sh
EXPOSE 9092
ENTRYPOINT ["/start-kafka.sh"]
CMD ["/opt/kafka/conf/server.properties"]
```

Como se pode ver pela Dockerfile anterior partimos de um CentOS na versão 6.8 ao qual é instalado o Java e o JDK através do *software package manager* YUM. Após isso é instalado o RPM do kafka na versão 0.10.0.1. Depois é copiado para a imagem o *script* de arranque do Kafka que neste caso é o ficheiro `start-kafka.sh` e para tornar este *script* executável é executado o comando `chmod +x /start-kafka.sh`. Para que outros contentores possam aceder ao Kafka é exposta a porta 9092 que é a sua porta oficial. O ponto de arranque é o *script* copiado anteriormente `start-kafka.sh` e a localização do ficheiro de configurações (`/opt/kafka/conf/server.properties`) é passado como argumento no momento de arranque.

Apache Zookeeper

A imagem do Apache Zookeeper é criada à custa da seguinte Dockerfile:

```
FROM repo-docker.example.com/centos6:6.8
ENV ZK_USER=zookeeper \
```

```
    ZK_DATA_DIR=/var/opt/zookeeper/data
RUN yum -y install env-java1.8 && \
    yum -y install jdk-ptin
RUN yum -y install zookeeper-3.4.9
COPY startZK.sh zkGenConfig.sh zkOk.sh zkMetrics.sh /opt/zookeeper/bin/
VOLUME ["/var/opt/zookeeper/data"]
EXPOSE 2181 3888 2888
ENTRYPOINT ["/opt/zookeeper/bin/startZK.sh"]
```

A construção da imagem do Zookeeper segue a abordagem que tem sido utilizada nas imagens anteriores. Como se pode ver pela sua Dockerfile esta tem como ponto inicial outra imagem baseada em CentOS na versão 6.8. De seguida são criadas duas variáveis de ambiente que são `ZK_USER` e `ZK_DATA_DIR`. A primeira permite definir qual o utilizador do Zookeeper e a segunda o diretório onde os dados a persistir devem ser guardados. Depois é instalado o Java e o JDK para que possa ser instalado o RPM do Zookeeper na versão 3.4.9. Para que no arranque da imagem existam os ficheiros de configuração estes devem ser disponibilizados no seu conteúdo. Para isso são copiados os seguintes ficheiros `startZK.sh`, `zkGenConfig.sh`, `zkOk.sh` e `zkMetrics.sh` para o diretório `/opt/zookeeper/bin/`. Para que os dados possam ser armazenados é criado um volume no diretório `/var/opt/zookeeper/data`. Por fim é definido o ponto de arranque da imagem que neste caso é `/opt/zookeeper/bin/startZK.sh` cujo o seu conteúdo é mostrado a seguir:

```
#!/usr/bin/env bash
/opt/zookeeper/bin/zkGenConfig.sh
/opt/zookeeper/bin/zkServer.sh start-foreground $@
```

Redis

A imagem do Redis é criada à custa da seguinte Dockerfile:

```
FROM repo-docker.example.com/centos6:6.8
RUN yum -y install redis-3.2.3
COPY redis-master.conf /etc/redis-master.conf
COPY redis-slave.conf /etc/redis-slave.conf
COPY run.sh /run.sh
VOLUME ["/var/lib/redis/master"]
VOLUME ["/var/lib/redis/slave"]
EXPOSE 6379
ENTRYPOINT [ "bash", "-c" ]
CMD [ "/run.sh" ]
```

Na imagem do Redis o seu ponto de partida é a imagem base do CentOS 6 na versão 6.8. Após isso é instalado o RPM do Redis na versão 3.2.3. Para que na imagem conste os ficheiros de configuração que permitem arrancar o Redis de forma conveniente são copiados os ficheiros `redis-master.conf` e o `redis-slave.conf` para o diretório `\etc`. Para além desses ficheiros também é copiado o ficheiro `run.sh` para a raiz "\" que vai ser a *script* de arranque do Redis. Depois são criados dois Volumes que vão albergar os diretórios de trabalho

do Redis, que são `/var/lib/redis/master` e o `/var/lib/redis/slave`. O serviço Redis é exposto na sua porta *standard* que é a 6379.

PostgresSQL

A imagem do sistema de gestão de base de dados relacional PostgreSQL é criada à custa da seguinte Dockerfile:

```
FROM repo-docker.example.com/centos6:6.8
RUN yum -y install env-postgresql9.4 \
    && yum -y install postgresql-server
ENV PGDATA /var/lib/pgsql/9.4/data
ENV PGPORT 5432
COPY docker-entrypoint.sh /usr/local/bin/
RUN chmod 744 /usr/local/bin/docker-entrypoint.sh
EXPOSE 5432
ENTRYPOINT ["/usr/local/bin/docker-entrypoint.sh"]
```

A imagem do PostgreSQL parte de uma imagem base do CentOS na versão 6.8, tal e qual como nas outras imagens mostradas anteriormente. De seguida é instalado os RPMs do PostgreSQL na versão 9.4. Depois são criadas duas variáveis de ambiente `PGDATA` e a `PGPORT`, ao qual a primeira define o diretório onde vão ser guardados os dados do PostgreSQL e a segunda variável define qual a porta onde o serviço será exposto. Depois é copiado para dentro da imagem o *script* que vai permitir o arranque do contentor que é o `docker-entrypoint.sh` para o diretório `/usr/local/bin/`. Para que esse *script* possa ser executado é dado permissões ao ficheiro através da execução do comando `chmod 744 /usr/local/bin/docker-entrypoint.sh`. Para que o serviço seja acedido através da rede é exposta a porta 5432 que é a *standard* no caso do PostgreSQL.

5.5 Ambiente de Testes

Após as imagens terem sido criadas e colocadas no repositório de imagens passam a estar disponíveis na rede e portanto pode-se iniciar o *deployment* das mesmas. Para que seja possível realizar o *deployment* é necessário criar um *cluster* Kubernetes em alta disponibilidade tal e qual como foi falado na Secção 4.6. Como se pode ver pela Figura 4.6 este *cluster* é constituído por 3 máquinas virtuais que são lançadas pelo comando pcloud que interage com o OpenStack. Cada uma das máquinas tem 4 VCPUs, 8 GB de RAM e 50 GB de Disco.

O Kubernetes instalado no *cluster* foi o da versão 1.7 que é última versão lançada até à data. A versão do Docker instalada foi a versão 1.12 que é a versão recomendada e suportada pelo Kubernetes [144].

5.6 *Deployment* da Aplicação

Após termos criados as imagens Docker e de termos um *cluster* Kubernetes funcional é necessário começar a realizar o *deployment* dos contentores da SmartIoT.

Como foi falado na secção 3.4.3, o Kubernetes para realizar o *deployment* de aplicações através da sua linha de comandos `kubectl` utiliza ficheiros de configuração em formato YAML ou JSON. Nestes ficheiros são especificados campos como por exemplo a imagem dos contentores, os volumes que devem ser associados, as portas que devem ser expostas, entre outros elementos configuráveis de um contentor.

Nesta secção vamos mostrar quais as configurações que foram criadas para cada um dos serviços que compõe a SmartIoT que vão permitir realizar o *deployment* dos vários contentores no *cluster* Kubernetes falado na secção anterior.

5.6.1 Cenário Esperado

É esperado um cenário de *deployment* da SmartIoT em que existe pelo menos duas instâncias dos serviços *stateless* em execução. Isto é feito para no caso de uma das instâncias falhar existir outra a fornecer serviço enquanto o Kubernetes relança o contentor falhado. No caso dos serviços *stateful* o seu *deployment* será no formato de *cluster* consoante a sua arquitetura.

Isto vai permitir que exista alta disponibilidade e balanceamento de carga pelas várias instâncias. Essa disposição é organizada na Figura 5.4.

Como para cada serviço existe mais do que uma instância a sua distribuição pelos nós deverá ser dispersa. Isto é, não existirá nenhum serviço em que todas as suas instâncias correm no mesmo nó do *cluster* Kubernetes. A distribuição das instâncias é feita desta forma para que no caso em que haja uma falha num dos nós do *cluster* Kubernetes o serviço continue a estar disponível.

5.6.2 *Deployment* dos Serviços Core

Os serviços que vamos realizar *deployment* previamente são os serviços *core* da SmartIoT e aqueles que o seu arranque não depende de outros contentores. Esses serviços são: Config Server, PostgresSQL, Apache Zookeeper, Apache Kafka e o Redis.

Para cada contentor e tendo em conta as características e os requisitos do serviço que fornece foi avaliado qual seria o melhor objeto do Kubernetes (abordados na Secção 3.4.3) para a realização do *deployment* e o seu controlo depois do contentor já se encontrar em execução. Para serviços que são criados em *cluster* em que cada um dos elementos do *cluster* tem de conhecer os seus pares como é o caso do Apache Kafka e do Apache Zookeeper foi escolhido o objeto `StatefulSet`. Este objeto foi escolhido para estes contentores porque permite obter

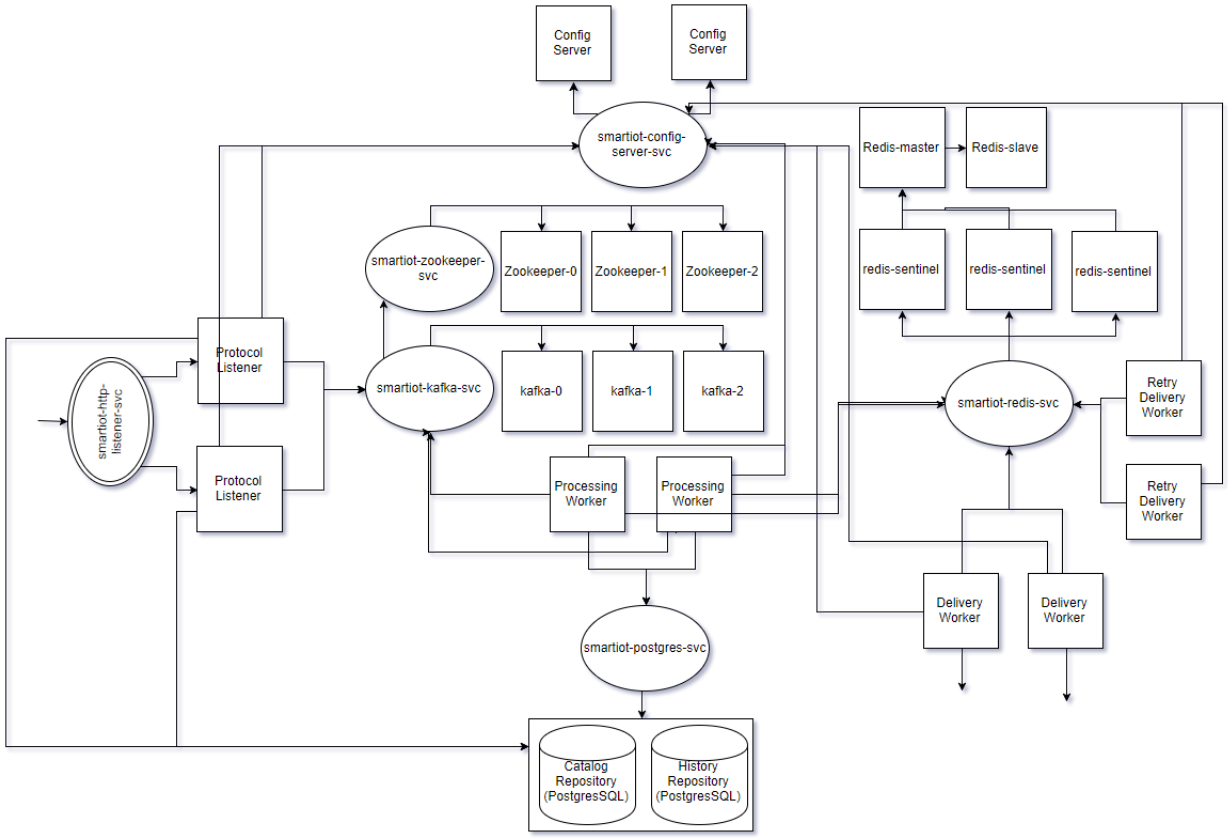


Figura 5.4: Organização lógica dos serviços que compõem a Smartiot em alta disponibilidade e balanceamento de carga.

identificadores de rede únicos para cada contentor. Ou seja, o serviço de nomes do Kubernetes consegue resolver o nome de cada um dos contentores individualmente, ao contrario do serviço *standard* do Kubernetes que redireciona e balanceia para um conjunto de Pods iguais, como foi abordado na Secção anterior 3.4.3.

Como o PostgreSQL não vai ser criado em *cluster* foi utilizado o objeto ReplicaSet que não permite a realização de atualizações por processos automáticos. A razão pela qual o *deployment* do PostgreSQL não é criado em *cluster* é fornecida na secção seguinte onde é abordado o *deployment* do PostgreSQL. Para o Redis como foi empregue outra estratégia de inicialização foi utilizado o objeto `Deployment` do Kubernetes. Este objeto foi escolhido devido aos seus parâmetros serem dinâmicos permitindo realizar *updates* automáticos sem perdas de serviço no caso de ser feito *upgrade* à versão da imagem.

Config Server

Para realizar o *deployment* do Config Server são utilizados dois tipos de objetos do Kubernetes que são o Service e o Deployment.

O Service é criado para que os outros contentores consigam aceder às configurações do Config Server. Para isso é criado um serviço interno ao *cluster* do tipo `ClusterIP` ao qual todos os

pedidos para a porta 9090 deste Service serão redirecionados para os contentores corretos. Como se pode ver pelo excerto do ficheiro de *deployment* imediatamente abaixo, este serviço vai se chamar `smartiot-config-server-svc` e será o endereço possível de resolver pelo *add-on* DNS do Kubernetes.

```
apiVersion: v1
kind: Service
metadata:
  name: smartiot-config-server-svc
spec:
  selector:
    app: smartiot
    svc: config-server
    env: dev
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 9090
      targetPort: 9090
```

O objeto Deployment é aquele que vai realizar o *deployment* propriamente dito dos contentores. Vão ser lançadas 2 réplicas da imagem `repo-docker.example.com/smartiot-config-server:1.1.0`, esperando com isso ter em todos os momentos 2 contentores a fornecer o serviço do Config Server. Esta ação de lançar 2 contentores deve-se maioritariamente por motivos de alta disponibilidade e também para permitir o balanceamento de carga.

Para que o Config Server possa disponibilizar as configurações a outros contentores este obviamente terá de ter acesso a essas configurações. Para isso é criado um ConfigMap que irá montar um volume dentro do contentor no diretório `/opt/ptin/smartiot/conf` com o respetivo ficheiro de configuração. Um exemplo do ficheiro de configuração que será disponibilizado é o que está no Anexo B.

A realização do *deployment* do Config Server descrito anteriormente é feita à custa do seguinte excerto YAML:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: smartiot-config-server-deploy
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: smartiot
        svc: config-server
        env: dev
```

```
    spec:
      containers:
      - name: smartiot-config-server
        image: repo-docker.example.com/smartiot-config-server:1.1.0
        ports:
        - containerPort: 9090
        volumeMounts:
        - name: config-volume
          mountPath: /opt/ptin/smartiot/conf
      volumes:
        - name: config-volume
          configMap:
            name: config-server-conf
```

PostgreSQL

À semelhança do Config Server falado anteriormente o *deployment* da base da dados PostgreSQL necessita de ter o objeto Kubernetes Service para que o serviço de base de dados esteja disponível aos outros Pods. Como se pode ver pelo excerto YAML imediatamente a seguir, o acesso à base dados é feito através do endereço `smartiot-postgres-svc` na porta *standard* 5432. Para isso é criado um serviço interno ao *cluster* do tipo `ClusterIP` que vai selecionar todos os contentores que fazem *match* com as seguintes Labels: "app: smartiot", "svc: postgres" e "env: dev", ao qual este conceito foi falado na Secção 3.4.3.

```
apiVersion: v1
kind: Service
metadata:
  name: smartiot-postgres-svc
spec:
  selector:
    app: smartiot
    svc: postgres
    env: dev
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 5432
```

Como estamos a realizar o deployment de um SGBD relacional com transações que tem de ser Atómicas, Consistentes, Isoladas e Duráveis (ACID) não podemos ter mais do que um contentor a lidar com esses dados. No caso de falha desse contentor a aplicação deve estar preparada para esse evento de falha e aguardar que o Kubernetes lance uma nova instância do PostgreSQL até voltar novamente a realizar as *queries* à base de dados. Para além disso as atualizações à imagem do PostgreSQL tem de ser feita com uma certa cautela num processo manual e num ambiente controlado sem serviços em execução para evitar que dados possam ser perdidos ou corrompidos. Devido a isso o *deployment* do PostgreSQL apenas terá de ter uma réplica em execução e para isso utilizamos o objeto ReplicaSet que ao contrário do objeto Deployment não realiza *updates* automáticos. Quanto à localização dos dados as transações são realizadas para

um Volume Persistente que é montado dentro do Pod no diretório `/var/lib/pgsql/9.4`. Este *deployment* é configurado através do excerto YAML a seguir:

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: smartiot-postgres-deploy
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: smartiot
        svc: postgres
        env: dev
    spec:
      containers:
      - name: smartiot-postgres
        image: repo-docker.example.com/postgresql:9.4
        ports:
        - containerPort: 5432
          name: pg-port-sec
        volumeMounts:
        - name: smartiot-postgres-pv
          mountPath: /var/lib/pgsql/9.4
      volumes:
        - name: smartiot-postgres-pv
          persistentVolumeClaim:
            claimName: smartiot-postgres-pv-claim
```

Apache Zookeeper

O *deployment* do Apache Zookeeper vai utilizar um objeto do Kubernetes que ainda não tinha sido utilizado antes que é StatefulSet. Como foi falado na Secção 3.4.3, este objeto reúne um conjunto de características que agilizam o *deployment* de serviços criados em *cluster* como é o caso do Apache Zookeeper. Ao utilizarmos este objeto é necessário criar dois tipos de Services. Um deles é o Service genérico do Kubernetes que é necessário para que os outros Pods do Smartiot mais propriamente o Apache Kafka possa aceder ao Zookeeper e utilizar este sistema na porta 2181, como pode ser visto no seguinte excerto:

```
apiVersion: v1
kind: Service
metadata:
  name: smartiot-zookeeper-svc
spec:
  selector:
    app: smartiot
    svc: zookeeper
    env: dev
  type: ClusterIP
```

```
  ports:
    - protocol: TCP
      port: 2181
```

O outro Service que é necessário criar é aquele que é utilizado pelos membros do cluster Zookeeper poderem comunicar uns com os outros. Este tipo de serviço na nomenclatura do Kubernetes é chamado de "*Headless Service*" e foi falado em mais detalhe anteriormente, na secção 3.4.3. Para que os membros do *cluster* possam comunicar com os seus pares (por exemplo para eleger o líder) é utilizado duas portas extras, para além da porta 2181, que é a porta 3888 e a porta 2888. A porta 3888 é utilizada para eleição do líder e a porta 2888 é utilizada para comunicação entre os membros do cluster. Para que isso aconteça é utilizado o seguinte excerto YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: smartiot-zookeeper-hsvc
spec:
  selector:
    app: smartiot
    svc: zookeeper
    env: dev
  ports:
    - port: 3888
      name: leader-election
    - port: 2888
      name: server
  clusterIP: None
```

O Zookeeper tem acesso a dois volumes utilizados em âmbitos diferentes. O volume com o nome `datadir` é um Persistent Volume e será o volume onde os dados permanecerão. O outro volume chamado `config-volume` é um volume criado para servir como ponto de acesso aos ficheiros de configuração do ConfigMap com o nome `zookeeper-conf`. Estes dois volumes são configurados pelo seguinte excerto YAML:

```
...
volumeMounts:
        - name: datadir
          mountPath: /var/opt/zookeeper/data
        - name: config-volume
          mountPath: /opt/zookeeper/conf
      volumes:
        - name: config-volume
          configMap:
            name: zookeeper-conf
  volumeClaimTemplates:
  - metadata:
      name: datadir
```

```
    annotations:
      volume.beta.kubernetes.io/storage-class: slow
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 1Gi
```

Apache Kafka

O *deployment* do Apacha Kafka em *cluster* é muito semelhante ao *deployment* do Zookeeper abordado anteriormente. Inicialmente é criado um *Service* do tipo *Headless Service* que permite comunicação direta com cada um dos membros do *cluster* Kafka. Para o *deployment* propriamente dito é utilizado o objeto StatefulSet que cria 3 réplicas do Kafka. Para que haja persistência de dados é utilizado três *Persistent Volumes* que são montados cada um deles em cada uma das réplicas do Kafka.

Redis

O *deployment* do Redis é feito à custa do exemplo criado pelo Kubernetes e disponível no GitHub em [145]. Este exemplo faz o *deployment* do Redis em *cluster* composto por um *Master* e múltiplos *Slaves*. Para além disso também faz o *deployment* de três Redis Sentinelas que fornecem alta disponibilidade e monitoria ao Redis [146].

Inicialmente é lançado um Pod com dois contentores, um Master e um Sentinela. Este Pod inicial é necessário para que no arranque do primeiro Sentinela este encontre o Master. Como o Master e o Sentinela estão no mesmo Pod permite que o Sentinela encontre o Master via *localhost*.

Para lidar com falhas relativas ao Redis Master é lançado um Redis Slave. O Redis Slave vai ter o papel de *backup* com replicação pois no caso de falha do Redis Master o Redis Slave assume o papel de Master.

Para lançar os Redis Slaves e os Sentinelas para além do Pod inicial foi utilizado o objeto Deployment do Kubernetes. No caso do Redis Master/Slave foi especificado 2 réplicas o que faz com que no momento do *deployment* o Kubernetes detete que já existe uma instância e apenas lance mais uma. Da mesma forma acontece com os Sentinelas, foi especificado 3 réplicas e no momento do *deployment* é detetado que já existe um contentor com um Sentinela pertencente ao Pod inicial e então são lançados apenas mais 2 Sentinelas.

Como foi dito, este *deployment* foi baseado no exemplo oficial criado pelo Kubernetes para o *deployment* do Redis. Neste exemplo o objeto que era sugerido para o *deployment* era o Replication Controller. Mas de acordo com a documentação mais recente do Kubernetes é recomendado utilizar o objeto Deployment em vez do objeto Replication Controller e devido a isso fez-se a alteração dos objetos [147].

Para que os outros contentores do Smartiot possam aceder ao Redis é criado o Service `smartiot-redis-svc` à escuta na sua porta oficial 26379.

5.6.3 *Deployment* dos Serviços *Stateless*

Os serviços *stateless*, mais propriamente o Protocol Listener, Processing Worker, Delivery Worker e Retry Delivery Worker, devido à sua simplicidade têm um *deployment* simples e com muitas semelhanças. Para o *deployment* destes serviços foi utilizado o objeto Deployment do Kubernetes.

De seguida é mostrado o ficheiro YAML com as configurações genéricas dos *deployments* dos vários serviços:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: smartiot-<NOME_SERVICO>-deploy
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: smartiot
        svc: <NOME_SERVICO>
        env: dev
    spec:
      containers:
      - name: smartiot-<NOME_SERVICO>
        image: repo-docker.example.com/<IMAGEM_SERVICO>:<VERSAO>
        args: ["-c http://smartiot-config-server-svc.default:9090/v1"]
        ports:
        - containerPort: <PORTA>
```

Como podemos ver para todos os casos é realizado o *deployment* de 2 réplicas para eliminar o ponto único de falha. Para obter as configurações iniciais é dado o argumento "`-c http://smartiot-config-server-svc.default:9090/v1`" que permite que cada um dos contentores se possa ligar ao ConfigServer.

Após o *deployment* estar definido é necessário criar o Service exterior para Protocol Listener, de maneira a que os utilizadores possam aceder à aplicação Smartiot do exterior. Devido a isso criou-se um Service do tipo `NodePort` disponibilizado na porta 30300 como podemos verificar pelo seguinte excerto YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: smartiot-http-listener-svc
spec:
  selector:
    app: smartiot
    svc: http-listener
    env: dev
  type: NodePort
```

```
ports:
  - protocol: TCP
    port: 3000
    nodePort: 30300
```

5.7 Avaliação

Como falado no início deste Capítulo na Secção 5.1, neste estudo de caso pretendemos avaliar através da ferramenta SmartIoT as seguintes características da plataforma Kubernetes:

1. *Deployment*;
2. Descoberta de serviços;
3. Recuperação de falhas;
4. Persistência de dados;
5. Configurações da aplicação;
6. Elasticidade;
7. Atualizações;

Após o *deployment* e o arranque de todos os contentores do SmartIoT foi possível verificar que os pontos 2 e 5 listados anteriormente tinham sido realizados com sucesso pois caso contrário a aplicação SmartIoT não estaria funcional. Pois sem o sistema de descoberta de serviços do Kubernetes e sem a introdução de configurações nos contentores através das ConfigMaps a SmartIoT não era capaz de ser iniciada. Então, nesta secção pretende-se avaliar os restantes pontos.

Para o ponto 1 vamos medir o tempo de *deployment* da aplicação SmartIoT para um ambiente Kubernetes e para um ambiente sem Kubernetes. Com isto espera-se provar a hipótese que o *deployment* sobre um ambiente Kubernetes é mais rápido do que o ambiente sem Kubernetes. Os resultados obtidos para este ponto são apresentados na Secção 5.7.1.

No ponto 3 pretendemos provar que o Kubernetes é uma ferramenta robusta capaz de lidar com várias falhas tanto ao nível do *cluster* como ao nível da aplicação que suporta. Para isso foram criados vários cenários com a intenção de introduzir falhas em vários elementos do *cluster* e desta maneira avaliar o impacto causado no serviço da aplicação SmartIoT e a forma como o Kubernetes reage a cada uma destas falhas. Os resultados obtidos para este ponto são apresentados na Secção 5.7.2.

No ponto 4 pretendemos provar que o Kubernetes é capaz de suportar serviços com persistência de dados e lidar com falhas a este nível sem colocar os dados em risco. Para isso foram introduzidas falhas no SGBD PostgreSQL em que os resultados obtidos são apresentados no Cenário 3 da Secção 5.7.2.

No ponto 6 pretendemos provar que o Kubernetes é um mecanismo que consegue escalar aplicações de uma forma mais rápida do que máquinas virtuais e com isso ser capaz de lidar

com a demanda de pedidos de uma forma mais ágil. Os resultados obtidos para este ponto são apresentados na Secção 5.7.3.

No ponto 7 pretendemos provar que o Kubernetes é capaz de realizar atualizações ao *software* da aplicação de uma forma simples e automatizada sem ser necessário parar o serviço fornecido aos clientes. Os resultados obtidos para este ponto são apresentados na Secção 5.7.4.

A ferramenta que permitiu perceber os impactos causados no estado do serviço da aplicação SmartIoT nos vários testes realizados foi o Vegeta [148]. O Vegeta é uma CLI que é capaz de realizar testes de carga HTTP com um ritmo de pedidos constante medindo a latência para cada pedido efetuado. Os valores da latência são disponibilizados em texto ou no formato de um gráfico.

5.7.1 *Deployment*

Com este teste pretende-se medir a quantidade de tempo necessária para realizar o *deployment* da aplicação SmartIoT conteinerizada sobre um *cluster* Kubernetes e o *deployment* da SmartIoT sem conteinerização sobre um *cluster* de máquinas virtuais sem Kubernetes.

Para realizar o *deployment* do SmartIoT sobre um *cluster* Kubernetes foi utilizado os ficheiros de configuração YAML criados na Secção 5.6. Para esta avaliação foram realizados três *deployments* em que o tempo médio de *deployment* foi de 1 minuto e 44 segundos.

Para realizar o *deployment* da SmartIoT não conteinerizada sobre um cluster de máquinas virtuais sem Kubernetes foram utilizados *playbooks* Ansible que foram criados antes da realização desta dissertação. Para esta avaliação foram realizados três *deployments* em que o tempo médio de *deployment* foi de 10 minutos e 33 segundos.

5.7.2 Recuperação de Falhas

Este teste pretende avaliar a forma como o Kubernetes lida com falhas e os efeitos que a recuperação dessas falhas podem trazer ao serviço. Para além disso, pretende-se saber o tempo que este necessita para realizar uma recuperação em vários tipos de falhas introduzidas nos elementos fundamentais do *cluster* Kubernetes, tais como:

- **Cenário 1:** Falha total de um nó;
- **Cenário 2:** Falha do processo kubelet num nó;
- **Cenário 3:** Falha do docker num nó com o PostgreSQL;
- **Cenário 4:** Falha do docker num nó sem o PostgreSQL;
- **Cenário 5:** Falha no processo do Protocol Listener.

Para cada cenário foi registado os seguintes tempos:

- O tempo em que o teste é iniciado;

- O tempo em que a falha é introduzida;
- O tempo em que o Kubernetes deteta a falha. É o momento em que o Kubernetes coloca o nó no estado "`Not Ready`";
- O tempo em que inicia as ações de recuperação.
- O tempo em que a aplicação fica num estado saudável. É o momento ao qual todos os contentores ficam no estado de "`Running`".

Em todos os cenários o Vegeta foi configurado para fazer pedidos a um ritmo constante de 100 pedidos por segundo.

Cenário 1

Neste cenário pretendemos investigar o que acontece ao serviço fornecido pelo SmartIoT quando um nó do *cluster* Kubernetes falha na totalidade. Para realizar esta avaliação foi selecionado o nó do *cluster* que no momento albergava mais contentores para inserir a falha. Os contentores que este nó albergava eram contentores pertencentes aos seguintes serviços da SmartIoT: Config Server, Protocol Listener, Processing Worker, Delivery Worker, Retry Delivery Worker, Kafka, Zookeeper e Redis. A este nó foi dada a ordem de *shut down* após 23 segundos do início do teste, o Kubernetes detetou a anomalia aos 63 segundos e iniciou a recuperação aos 70 segundos terminando aos 86 segundos.

Os resultados deste teste estão representados no gráfico da Figura 5.5. Neste gráfico foram acrescentados os tempos medidos ao longo do teste onde é possível verificar que o Kubernetes demorou cerca de 40 segundos a detetar a anomalia e cerca de 16 segundos a recuperar, relançando todos os contentores falhados noutro nó saudável. No total o Kubernetes necessitou de 1 minuto e 3 segundos para lidar com a falha introduzida neste cenário e nenhuma manobra de recuperação ou a falha total do nó levou à perda de serviço da SmartIoT.

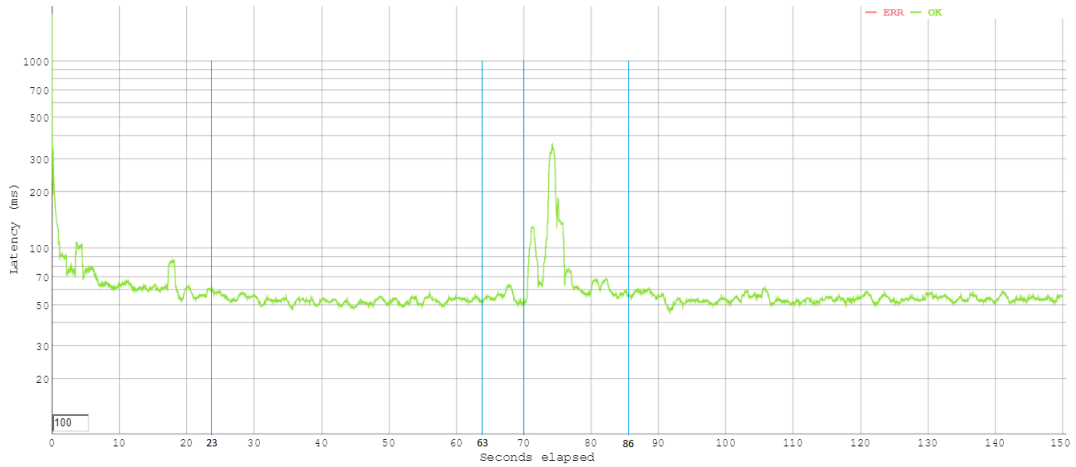


Figura 5.5: Gráfico com a latência do serviço que representa o impacto causado pela falha de um dos nós do *cluster*. As linhas verticais azuis representam os tempos registados no decorrer da experiência.

Cenário 2

Neste cenário pretendemos avaliar os impactos que uma falha no processo kubelet de um dos nós pode causar ao serviço da SmartIoT. Esta falha foi introduzida através do comando *kill* que matou o processo kubelet num nó previamente selecionado. O nó selecionado para este cenário de avaliação tinha os seguintes serviços da SmartIoT: Config Server, Protocol Listener, Processing Worker, Delivery Worker, Retry Delivery Worker, Kafka, Zookeeper e Redis.

Os valores obtidos desta avaliação podem ser vistos no gráfico da Figura 5.6. Neste gráfico estão identificados os seguintes tempos: aos 14 segundos foi parado o processo kubelet, aos 61 segundos o Kubernetes detetou a falha, aos 320 segundos iniciou a recuperação e aos 336 segundos tinha todos os contentores relançados noutro nó num estado de *Running*.

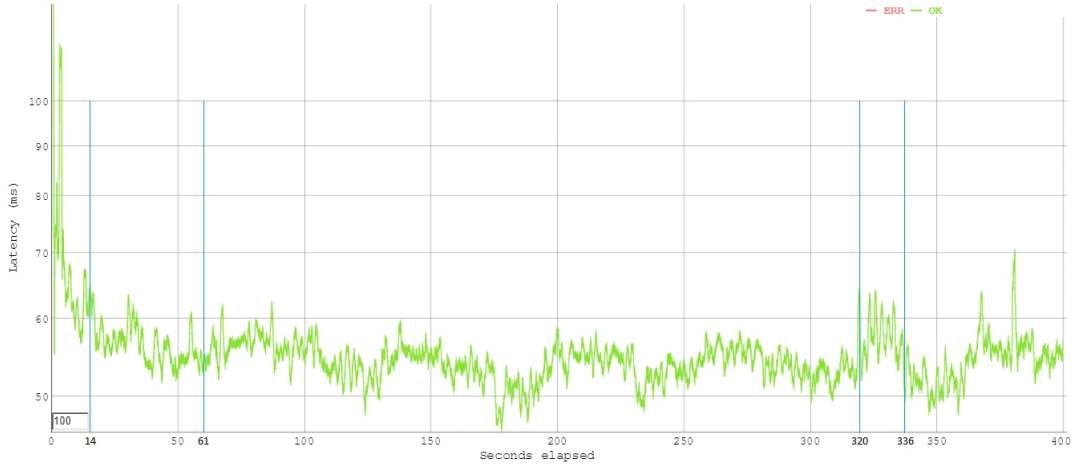


Figura 5.6: Gráfico com a latência do serviço que representa o impacto causado pela falha do processo kubelet num dos nós do *cluster*. As linhas verticais azuis representam os tempos registados no decorrer da experiência.

Nesta avaliação percebemos que o Kubernetes demorou cerca de 47 segundos a detetar a falha do kubelet. Como esta falha poderia ser provocada por instabilidades ao nível da rede que impedia a comunicação com o serviço kubelet, o Kubernetes iniciou um tempo de análise mais longo que foi cerca de 4 minutos e 19 segundos até concluir que o nó estava efetivamente com problemas. A recuperação e o relançamento dos contentores afetados num nó saudável do *cluster* demorou cerca de 16 segundos. Neste cenário todos os serviços afetados foram recuperados e não houve falha no serviço da SmartIoT.

Cenário 3

Com o cenário 3 pretende-se avaliar os impactos causados ao serviço da SmartIoT e a sua recuperação quando envolve persistência de dados. Para isso foi parado o Docker, através do comando *kill* num nó do *cluster* que tem o contentor do PostgreSQL em execução. Como foi abordado na Secção 5.6.2, o PostgreSQL foi o único elemento da SmartIoT que não foi replicado por vários contentores devido a possíveis problemas que poderiam causar inconsistência aos dados armazenados. Devido a este fator espera-se que a partir do momento em que o contentor do PostgreSQL falhe o serviço da SmartIoT seja interrompido até ao fim da recuperação. Para

além disso, como o PostgreSQL utiliza um Persistent Volume (conceito falado na secção 3.4.3.2) pretende-se medir o tempo que demora a interação entre o Kubernetes e o OpenStack que vai permitir a deslocação do Volume para outro nó do *cluster* Kubernetes. Então para a realização da avaliação pretendida neste cenário foi parado o processo do Docker num dos nós do *cluster*. Este nó do *cluster* albergava os seguintes contentores: PostgreSQL, Config Server, Protocol Listener, Processing Worker, Delivery Worker, Retry Delivery Worker, Kafka, Zookeeper e Redis.

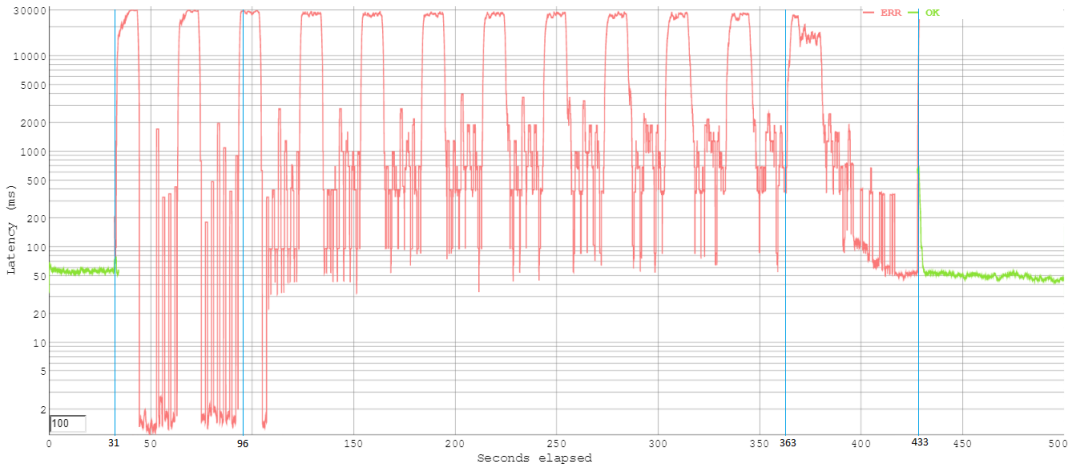


Figura 5.7: Gráfico com a latência do serviço que representa o impacto causado pela falha do processo docker num dos nós do *cluster* que continha a base de dados PostgreSQL. As linhas verticais azuis representam os tempos registados no decorrer da experiência.

Como se pode ver pela Figura 5.7 o Kubernetes demorou cerca de 65 segundos para detetar a falha. Da mesma forma que no cenário anterior, esta falha poderia ser um problema de rede que estava a impedir a comunicação com os contentores o que levou que o Kubernetes demorasse cerca de 4 minutos e 27 segundos a dar o nó como perdido. Após isso demorou 1 minuto e 10 segundos a recuperar a falha, relançando todos os contentores falhados incluindo o PostgreSQL noutro nó do *cluster*. Como era esperado, entre o momento em que o erro é iniciado e os contentores afetados são relançados noutro nó do *cluster* o serviço da SmartIoT falhou, mas após isso recuperou passando a fornecer o serviço novamente.

Cenário 4

Este cenário pretende avaliar os efeitos causados no serviço da aplicação SmartIoT no momento em que o processo do Docker falha num dos nós do *cluster*. A falha no Docker foi introduzida através do comando *kill* que matou o processo do Docker. No cenário 3 esta avaliação já foi realizada em parte, mas como o SGBD PostgreSQL fazia parte da lista de contentores afetados o serviço da SmartIoT ficou indisponível durante a falha, como já era previsto. Neste cenário pretendemos realizar a mesma avaliação mas para o caso em que o contentor do PostgreSQL não é afetado. O nó selecionado albergava os seguintes contentores: Config Server, Protocol Listener, Processing Worker, Delivery Worker, Retry Delivery Worker, Kafka, Zookeeper e Redis.

Nesta avaliação foram registados os seguintes tempos: aos 64 segundos foi parado o processo

Docker, aos 114 segundos o Kubernetes detetou a falha, aos 377 segundos o Kubernetes iniciou o relançamento dos contentores afetados e aos 385 segundos todos os contentores estavam num estado de "*Running*".

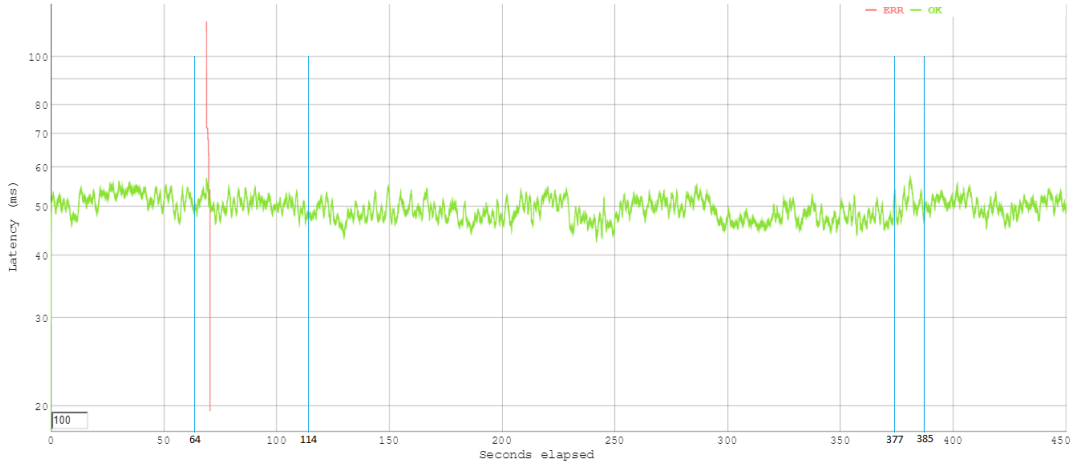


Figura 5.8: Gráfico com a latência do serviço que representa o impacto causado pela falha do processo Docker num dos nós do *cluster* Kubernetes. As linhas verticais azuis representam os tempos registados no decorrer da experiência.

Desta avaliação resultou o gráfico da Figura 5.8 onde é possível ver que o Kubernetes demorou cerca de 50 segundos a detetar a anomalia, depois iniciou um tempo de espera de cerca de 4 minutos e 23 segundos para dar o nó como perdido (pelas razões descritas no cenário anterior) e após isso relançou os contentores afetados noutro nó do *cluster* em 8 segundos. Como se pode ver pelo gráfico, após o momento em que o Docker é parado existe uma quantidade de pedidos que ficam sem resposta. Estes eram os pedidos que estavam a ser processados no momento em que um dos contentores do Protocol Listener falhou devido à falha do processo do Docker.

Cenário 5

Neste cenário pretende-se introduzir falhas ao nível da aplicação para avaliar como o Kubernetes reage quando um dos processos da aplicação conteinerizada morre. Para isso, vamos matar o processo com PID 1 num dos contentores do Protocol Listener através do comando *kill*. Como estes contentores são aqueles que lidam em primeira instância com os pedidos do cliente conseguimos avaliar com mais eficácia se a perda de um destes contentores provocado por uma falha interna vai provocar a quebra do serviço da SmartIoT e se o *cluster* consegue recuperar essa perda.

Neste cenário o Vegeta foi configurado para fazer pedidos constantes a uma taxa de 200 pedidos por segundo. Com isto espera-se ver uma quantidade de pedidos não respondidos entre o momento em que o erro é introduzido e o momento em que o contentor é reiniciado.

Os resultados obtidos para esta avaliação encontram-se no gráfico da Figura 5.9. Como podemos ver pelo gráfico aos 60 segundos é dada a ordem de *kill* de um dos processos com o Protocol Listener. A partir desse momento existe uma quantidade de pedidos que não são

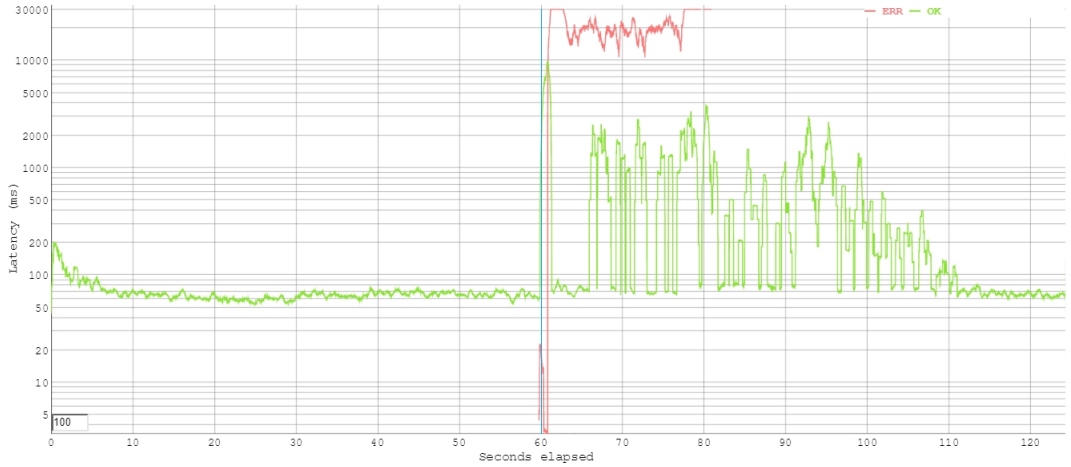


Figura 5.9: Gráfico com a latência do serviço que representa o impacto causado pela falha do processo do Protocol Listener.

respondidos (linhas do gráfico a vermelho), após 20 segundos um novo contentor é iniciado e todos os pedidos passam a ser respondidos. Progressivamente os valores da latência diminuem para valores próximos de 65 milissegundos, que era o valor da latência antes do erro. Tendo em conta os resultados obtidos o Kubernetes foi capaz de recuperar a falha.

5.7.3 Elasticidade

Nesta secção pretende-se avaliar os efeitos no serviço da SmartIoT ao serem despoletadas ações de escalabilidade horizontal (conceito abordado na Secção 2.4.2) do Kubernetes ao nível dos seus contentores. Para isso foi colocado apenas um contentor do elemento Protocol Listener da SmartIoT ativo de maneira a que este pudesse escalar para dar resposta a uma demanda elevada de pedidos. Para que fosse possível existir escalabilidade automática foi necessário associar o objeto Horizontal Pod Autoscaler ao Deployment do Protocol Listerner. Isso foi feito à custa do seguinte comando: `kubectl autoscale deployment smatiot-protocol-listener-deploy -min=1 -max=3 -cpu-percent=50`. Este comando cria um *autoscaler* para o *deployment* do Protocol Listener, com um *threshold* de 50% de CPU e um número de réplicas que pode variar entre 1 e 3.

Para aumentar a carga de CPU através de pedidos foi utilizado o Vegeta que foi configurado para realizar pedidos a um ritmo constante de 350 pedidos por segundo. O valor do ritmo de pedidos foi obtido através de um teste prévio (ver Anexo E) que permitiu obter os valores da taxa de pedidos respondidos em função da variação do número de réplicas do Protocol Listener. Esta quantidade de pedidos irá fazer com que a carga de CPU do Protocol Listener aumente e com isso despolete as ações de elasticidade.

Os valores da latência obtidos que representam o estado do serviço no decorrer desta avaliação estão representados no gráfico da Figura 5.10. Neste gráfico é possível ver que inicialmente existia uma quantidade de pedidos que não eram respondidos (valores assinalados com cor vermelha),

Figura 5.10: Gráfico com o impacto das ações de elasticidade no elemento Protocol Listener.

após o lançamento da primeira réplica do Protocol Listener próximo dos 60 segundos e após o arranque da mesma (30 segundos depois) todos os pedidos passaram a ser respondidos. Com a terceira réplica lançada aos 120 segundos os valores da latência foram diminuindo para próximo dos 50 milissegundos. A partir do gráfico é possível ver que o arranque completo do Protocol Listener acontece após 30 segundos do seu lançamento que é o momento em que todos os pedidos realizados pelo Vegeta passam a ser respondidos.

De maneira a comparar estes resultados com um ambiente baseado apenas em máquinas virtuais foram medidos o tempo necessário para arrancar um nova máquina virtual e o tempo de instalação e arranque do Protocol Listener. O tempo que demora arrancar uma máquina virtual é de 56 segundos e o Protocol Listener está disponível após 14 segundos.

5.7.4 Atualizações

Como foi falado na Secção 3.4.3.2, no momento de atualização da imagem de um contentor o Kubernetes inicia um processo controlado que permite realizar essa atualização sem perda de serviço. Então nesta secção pretende-se avaliar e medir os impactos causados pelo processo de atualização do Kubernetes na aplicação SmartIoT.

Para isso foi configurado o Vegeta para fazer pedidos a uma taxa de 50 pedidos por segundo durante 120 segundos. No decorrer dos 120 segundos foi lançado um novo *deployment* do serviço Protocol Listener com uma imagem numa nova versão. Os resultados obtidos podem ser consultados no gráfico da Figura 5.11.

Como se pode ver pelo gráfico da Figura 5.11 aos 65 segundos é dado o comando de *deployment* da nova imagem do Protocol Listener e após isso existe um ligeiro aumento da latência nos pedidos efetuados pela ferramenta Vegeta. Esse é o único impacto registado pois o serviço continuou em funcionamento. Aos 80 segundos e após 15 segundos da ordem de atualização o processo de atualização do Protocol Listener é concluído.

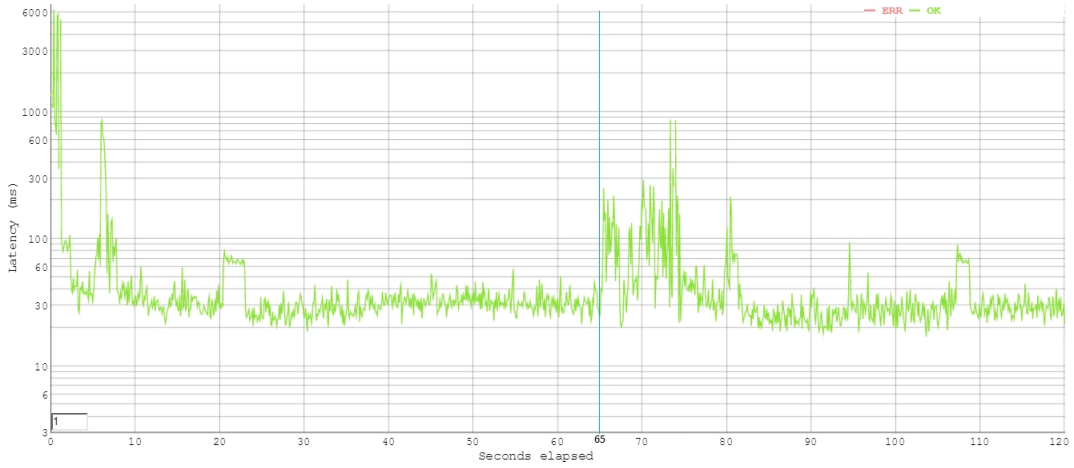


Figura 5.11: Gráfico com o impacto da atualização da imagem do elemento Protocol Listener.

Capítulo 6

Análise de Resultados

No Capítulo anterior realizamos um "Estudo de Caso" através da aplicação SmartIoT, no qual esta aplicação foi conteinerizada para que pudesse ser realizado o seu *deployment* sobre um *cluster* Kubernetes. Com isto foi possível realizar um conjunto de testes que colocaram o Kubernetes à prova em vários aspetos. Neste Capítulo vamos discutir os resultados obtidos dessa avaliação, expondo os benefícios que a conteinerização de aplicações pode trazer a uma organização.

Face à falta de trabalho científico produzido nestas áreas não foi possível comparar os resultados obtidos com outros trabalhos que permitissem auferir a qualidade dos mesmos.

6.1 *Deployment* de Aplicações

O *deployment* de aplicações é realizado tanto no ambiente de desenvolvimento como no ambiente de produção. No ambiente de desenvolvimento são realizados testes à aplicação à custa de vários *deployments* que vão permitir eliminar erros tornando aplicação robusta e apta para ser executada no ambiente de produção. Tendo em conta os resultados obtidos na Secção 5.7.1 relativos a avaliação do *deployment* da aplicação SmartIoT é possível concluir que o *deployment* de uma aplicação conteinerizada num ambiente Kubernetes é mais rápido do que o ambiente sem conteinerização e Kubernetes. Esta redução de tempo de *deployment* vai permitir que para cada conjunto de testes, em que seja necessário a realização de um *deployment* completo da aplicação, vamos demorar 6 vezes menos caso o *deployment* fosse realizado diretamente sobre máquinas virtuais. Transpondo esse valor para um contexto real, se numa aplicação forem realizados durante um dia cerca de 20 *commits* de alterações ao *source code* que despoletem cerca de 20 novos *deployments* no *cluster* de testes, de acordo com os dados obtidos na Secção 5.7.1, se estes *deployments* fossem realizados diretamente sobre máquinas virtuais iríamos necessitar de pelo menos 3 horas e 31 minutos, sem contabilizarmos o tempo dedicado à realização dos testes. Em comparação com um ambiente conteinerizado sobre Kubernetes apenas era necessário cerca de 34 minutos e 40 segundos para realizar a mesma quantidade de *deployments*.

Este resultado obtido também permite concluir que através da conteinerização estamos a

reduzir o tempo necessário de desenvolvimento de aplicações, fazendo com que os produtos de *software* ou novas funcionalidades cheguem mais rápido ao cliente.

6.2 Atualização de *Software*

Atualizar um *software* é algo que é necessário para manter este atrativo, com novas funcionalidades e mais seguro. A realização deste processo deve ser o mais rápida possível com um tempo de *down time* próximo de zero, dependendo da arquitetura da aplicação ou do elemento que estamos atualizar. De acordo com os resultados obtidos na Secção 5.7.4 foi possível verificar que uma atualização a uma aplicação conteinerizada sobre Kubernetes era realizada em 15 segundos, sem falhas no serviço, através de um processo simples e automático despoletado por um único comando.

Caso esta atualização fosse realizada num ambiente não conteinerizado e sem Kubernetes era necessário um processo mais complexo e demorado. Normalmente para realizar atualizações a aplicações que correm diretamente sobre máquinas virtuais (ou *bare metal*) o processo de atualização é o seguinte: cria-se uma nova máquina para que se possa nesta instalar e configurar a versão do *software* atualizado, após isso é necessário alterar as configurações do balanceador de carga de maneira desviar o tráfego para esta nova máquina e por fim eliminar a máquina com a versão do *software* antigo. Pelos resultados obtidos na Secção 5.7.3, relativos ao tempo de arranque de uma máquina virtual, era necessário 56 segundos para arrancar uma única máquina virtual. Se ao tempo da criação da máquina virtual adicionarmos o tempo de preparação do teste e das configurações necessárias ao balanceador de maneira a desviar o tráfego ultrapassamos em muito o tempo necessário para a realização de uma atualização a uma aplicação conteinerizada. Para além disso, devido ao processo de atualização de contentores realizado pelo Kubernetes (abordado na Secção 3.4.3.2) não é necessário adicionar uma nova máquina ao *cluster* para evitar que durante a atualização o serviço falhe, tornando esta opção mais eficiente do ponto de vista dos recursos utilizados.

Desta forma podemos concluir que as atualizações ao *software* realizadas num ambiente conteinerizado sobre Kubernetes são mais eficientes, rápidas e simples de serem realizadas.

6.3 Elasticidade

Uma aplicação que assenta sobre uma infraestrutura elástica consegue lidar mais facilmente com um fluxo inconstante de pedidos por parte dos clientes de uma forma mais eficiente. Esta eficiência advém do facto de a cada momento apenas são alocados os recursos necessários capazes de dividir o tráfego por várias instâncias e dar resposta aos pedidos dos clientes. Em aplicações que são instaladas diretamente sobre máquinas virtuais a unidade de escalabilidade é a máquina virtual. De acordo com os resultados obtidos das avaliações realizadas anteriormente na Secção 5.7.3, uma máquina virtual consegue se iniciar em cerca de 56 segundos. Ao utilizarmos conteinerização

o valor de arranque obtido para um contentor (ou Pod no caso do Kubernetes) é feito em cerca de 15 segundos. Ao compararmos estes dois valores é possível concluir que com a utilização de contentores é possível obter escalabilidade de uma maneira mais rápida. Para além disso com a utilização de apenas máquinas virtuais estamos acrescentar um *overhead* ao sistema, pois por cada máquina virtual temos um sistema operativo completo, como foi abordado na Secção 2.2. Outro ponto positivo a considerar com a utilização do Kubernetes é que não é necessário implementar o sistema que gere as regras do balanceador de carga que desvia o tráfego pelas novas instâncias criadas por atividades de escalabilidade automática. Esta funcionalidade é feita pelo elemento kube-proxy do Kubernetes (elemento abordado na Secção 3.4.3.4).

Estas atividades de escalabilidade horizontal em aplicações conteinerizadas podem esgotar os recursos do *cluster* Kubernetes levando a que seja necessário a intervenção humana para acrescentar novos nós ao *cluster*. Com a criação do comando asgroup abordado na Secção 4.6.3 esta realidade não é verdade. Pois com a utilização dos grupos de escalabilidade automática à medida que novos Pods vão sendo acrescentados aos nós de cluster, estes vão elevando a carga de CPU desses nós, fazendo com que despolete ações de escalabilidade geridas pelo comando pcloud asgroup, que vai iniciar novas máquinas virtuais que vão ser acrescentadas ao *cluster* Kubernetes. Com este sistema é possível concluir que temos um *cluster* totalmente elástico, tanto ao nível da escalabilidade horizontal dos contentores como dos nós do *cluster*, que fornece a cada momento a quantidade de recursos necessária para dar resposta às necessidades do momento, diminuindo a necessidade de intervenção manual no *cluster* por mais tempo e tornando o sistema mais autónomo e eficiente.

6.4 Plataforma Robusta

Um ponto avaliado no Capítulo anterior na Secção 5.7.2 foi a forma como o Kubernetes lida com a falha nalguns dos seus elementos assim como nas aplicações que suporta. Com esta avaliação foi possível demonstrar a robustez do Kubernetes na presença de falhas mantendo os serviços que suporta disponíveis. Esta avaliação foi dividida em cinco cenários que permitiu avaliar o Kubernetes quando existe a falha total de um nó do *cluster*, a falha do processo kubelet, a falha da plataforma de conteinerização Docker e a falha de um dos processos da aplicação. Em todos os testes realizados foi possível concluir que o Kubernetes conseguiu recuperar da falha relançando os contentores afetados noutro nó saudável e restabelecendo o serviço. Apenas no cenário que afetava o SGBD PostgreSQL é que o serviço ficou totalmente em baixo durante o período de recuperação. Como foi dito, este resultado já era esperado pois o *deployment* do PostgreSQL não foi feito em *cluster*, com redundância dos dados.

Com os resultados obtidos é possível concluir que o Kubernetes é uma plataforma bastante robusta que é capaz de lidar com falhas dos seus componentes assim como com falhas das próprias aplicações que suporta. A utilização do Kubernetes permite tornar as aplicações mais robustas a falhas e com uma capacidade de recuperação maior caso fossem instaladas diretamente sobre máquinas virtuais.

6.5 Disparidade entre o Ambiente de Desenvolvimento e Produção

A instalação e configuração de sistemas por processos manuais nas plataformas de Desenvolvimento e Produção levam em maior parte dos casos à criação de desigualdades entre estes dois ambientes. Para além disso, por vezes as plataformas físicas que suportam estes dois ambientes são diferentes. Estes dois aspetos levam a que exista um risco elevado do *deployment* de aplicações falhar no ambiente de produção.

Com a utilização da plataforma de conteinerização Docker que aumenta a portabilidade de uma aplicação (conceito abordado na Secção 3.3) e a utilização de uma *pipeline* de *Continuous Delivery* (CD) (como sugerido na Secção 4.1) é possivel reduzir a disparidade entre o ambiente de desenvolvimento e o ambiente de produção. Isto acontece porque o programador ao criar ficheiros de configuração YAML para a realização do *deployment* sobre o *cluster* Kubernetes de testes vai testar intensivamente o *deployment* da aplicação. Então, como o *cluster* de produção também é um cluster Kubernetes como o que é utilizado na fase de testes esses ficheiros YAML testados podem ser reutilizados, reduzindo os problemas associados ao *deployment* de uma aplicação em produção. Por consequência, os programadores ganham uma forma simples de ter acesso a um ambiente de *deployment* muito idêntico ao de produção fazendo com que as aplicações possam ser criadas desde as suas fases iniciais sobre este ambiente idêntico ao de produção. Para além disso, com esta estratégia, parte das responsabilidades das equipas de operação que têm a tarefa de realizar o *deployment* e manter o *cluster* em produção passa para as equipas de desenvolvimento que criam as configurações de *deployment* durante a fase de desenvolvimento, reduzindo o atrito entre estas duas equipas e criando um ponto de equilíbrio entre elas.

6.6 Redução da Utilização de Recursos

Quando comparamos os contentores com as máquinas virtuais na Secção 2.2 vimos que os contentores arquiteturalmente são uma opção mais eficiente à utilização de máquinas virtuais, pois não têm um sistema operativo completo dentro deles. Só por esta razão permitem reduzir globalmente a quantidade de recursos utilizada.

O *deployment* de uma aplicação não conteinerizada sobre um ambiente estático composto apenas por máquinas virtuais que não utilizam os mecanismos de escalabilidade horizontal do Kubernetes ou os grupos de escalabilidade automática (asgroups) adicionados à CLI pcloud faz com que este tipo de *deployment* tenha de ser realizado sobre uma infraestrutura estática com recursos suficientes capazes de suportar todas eventualidades possiveis. Com a introdução de conteinerização e dos mecanismos de elasticidade reduzimos a utilização de recursos, pois em cada momento a plataforma está sobre uma análise constante e sempre que seja necessário mais recursos esses recursos são provisionados à aplicação no momento necessário, como podemos ver pela avaliação da Secção 5.7.3.

Como vimos anteriormente na Secção 6.2 deste Capítulo, outro aspeto em que a conteinerização de aplicações e a utilização de Kubernetes permite reduzir a utilização de recursos é no momento da atualização da versão do *software*. Devido ao processo de atualização automático que o Kubernetes utiliza, falado na Secção 3.4.3.2, não existe a necessidade de lançar uma máquina virtual extra para instalar as atualizações sem quebras de serviço, estas são feitas nos contentores que são iniciados nas máquinas existentes que compõe o *cluster* Kubernetes.

Outro aspeto em que a conteinerização de aplicações permite reduzir a utilização de recursos é quando numa aplicação são utilizadas versões diferentes do mesmo *software*. Neste caso, sem a utilização de conteinerização, se existisse duas versões diferentes do mesmo *software* estas teriam de ser instaladas em duas máquinas virtuais. Com a utilização de conteinerização cada versão do *software* é disponibilizado em imagens Docker diferentes e essas duas imagens podem ser instanciadas em contentores numa única máquina.

Devido a estes fatores é possível concluir que a conteinerização de aplicações sobre Kubernetes permite reduzir a utilização de recursos.

6.7 Desenvolvimento de Aplicações Conteinerizadas

O desenvolvimento de aplicações conteinerizadas numa arquitetura de micro-serviços (conceito abordado na Secção 2.3) requer que exista um sistema de descoberta de serviços. Sem este sistema os serviços dispersos pelo *cluster* a correr dentro de contentores não conseguiam encontrar os outros serviços no momento de arranque. Com a utilização de Kubernetes este sistema já não necessita de ser implementado pois o Kubernetes dispõe deste serviço, como falado na Secção 3.4.3.2.

Outro aspeto importante que o Kubernetes vem trazer ao desenvolvimento de aplicações são os objetos ConfigMaps e Secrets, abordados na Secção 3.4.3.2. Estes permitem aumentar a portabilidade das aplicações conteinerizadas, pois caso contrario, para cada alteração realizada às configurações ou chaves era necessário criar uma nova imagem Docker.

Um dos grandes desafios à conteinerização de aplicações é as aplicações com estado, criadas por vezes em *cluster*. Como os contentores são uma unidade efémera que podem percorrer vários nós do *cluster* aumentam a complexidade da execução deste tipo de aplicações num ambiente conteinerizado e disperso. O Kubernetes com a utilização dos Persistent Volumes (conceito abordado na Secção 3.4.3.2) e dos StatefulSets (conceito abordado na Secção 3.4.3.1) permite reduzir esta complexidade. O Kubernetes ao disponibilizar volumes externos persistentes permite que o contentor possa ser relançado em nós diferentes do cluster, pois no seu arranque o mesmo volume com os dados persistentes será disponibilizado novamente, como foi demonstrado anteriormente no Cenário 3 da Secção 5.7.2. Com a utilização do objeto Statefulset é possível coordenar o *deployment* deste tipo de aplicações e para além disso torna mais fácil manter o seu estado nos sucessivos relançamentos, pois o endereço para aceder a cada um dos contentores mantém-se.

Devido a estes fatores é possível concluir que o Kubernetes facilita o desenvolvimento de aplicações conteinerizadas pois alguns dos problemas existentes na execução de aplicações conteinerizadas são solucionados pela plataforma Kubernetes.

6.8 Redução do Tempo de Desenvolvimento

Devido a fatores como a redução do tempo de *deployment*, as atualizações serem automáticas e mais rápidas, a redução da disparidade entre o ambiente de desenvolvimento e produção e a facilidade no desenvolvimento de aplicações conteinerizadas permitem reduzir o tempo do ciclo de desenvolvimento de *software* quando este é conteinerizado e executado sobre a gestão da plataforma Kubernetes.

6.9 Dificuldades

Durante este projeto foram sentidas algumas dificuldades essencialmente no momento de conteinerização de alguns sistemas com estado e na passagem de todos estes conhecimentos às equipas de desenvolvimento.

Alguns sistemas ainda não estão preparados para os conceitos e desafios da conteinerização. Por exemplo alguns sistemas distribuídos, em certas situações, identificam os seus pares através de um endereço IP e não de um nome que possa ser resolvido por DNS. Num ambiente Kubernetes o IP dos Pods devido à sua efemeridade pode mudar com frequência e nesse caso utiliza-se um nome para manter um estado permanente entre esse nome e o Pod correto. Como o conceito de conteinerização é bastante recente no desenvolvimento de *software* acreditamos que vários sistemas em breve vão se focar mais neste assunto tornando a sua conteinerização mais simples.

A conversão de um sistema criado numa arquitetura monolítica para uma arquitetura baseada em micro-serviços é complexa, fazendo com que seja necessário, por vezes, uma reestruturação total da aplicação.

Como vimos ao longo desta dissertação a conteinerização de aplicações e o *deployment* das mesmas sobre um ambiente Kubernetes requer um grande conhecimento que abrange várias áreas e conceitos. Passar todo este conhecimento de uma forma eficaz às equipas de desenvolvimento e operação de grandes dimensões será o maior desafio na adoção da conteinerização de aplicações.

Capítulo 7

Conclusões

Como vimos no Capítulo 1, o principal objetivo desta dissertação é estudar os conceitos da conteinerização de aplicações de maneira a propor um caminho que permita a uma organização adotar as novas tecnologias em torno da conteinerização e com isso beneficiar das vantagens que esses novos paradigmas lhes possam trazer. Para isso, no decorrer desta dissertação provou-se que estas tecnologias vão melhorar o desempenho de uma organização que desenvolve *software*. Este projeto foi desenvolvido na AlticeLabs que serviu de organização modelo onde todas as implementações e sugestões foram baseadas no contexto desta organização.

Inicialmente abordamos os conceitos fundamentais que reuniram uma base de conhecimento suficiente para que fosse possível compreender as soluções propostas nesta dissertação. Após reunirmos esta base de conhecimento fundamental passamos para o estudo das tecnologias que vão estar diretamente ligadas às soluções propostas. Nesta parte do texto, começamos por abordar a base tecnológica de todo o trabalho desta dissertação que é o sistema de operação de *clouds* privadas chamado OpenStack, onde abordamos a sua arquitetura e todos os seus componentes fundamentais. No próximo passo, para cumprir o objetivo de adicionar à CLI pcloud uma interface de gestão simples para o sistema de elasticidade de máquinas virtuais do OpenStack fomos estudar as interfaces de duas das *clouds* públicas mais populares (a Google Cloud Plataform e a Amazon AWS). Neste contexto, de seguida abordamos a CLI pcloud para entender a sua arquitetura e implementação existente, de maneira a realizar os desenvolvimentos criados no âmbito desta dissertação. Este sistema elástico permitiu aumentar a elasticidade de *clusters* compostos por máquinas virtuais, como é o caso do *cluster* Kubernetes. De seguida, ao nível tecnológico entramos no principal foco desta dissertação nas tecnologias de conteinerização Docker e Kubernetes. Relativamente a estas duas tecnologias foram abordados os seus conceitos, arquitetura e componentes principais.

Após a base teórica inicial estar concluída, iniciamos a parte de desenvolvimento. Na fase inicial do desenvolvimento foi criado um diagrama lógico dividido nas principais fases de desenvolvimento de *software*, as fases *Create*, *Build*, *Deploy* e *Run*. Este diagrama intercala os elementos e conceitos que envolvem o desenvolvimento de aplicações conteinerizadas, tais como contentores, imagens Docker, repositórios de imagens e o *deployment* dos contentores sobre o

ambiente Kubernetes nas *pipelines* de CI/CD. Esse diagrama criado serviu de base para o resto do estudo permitindo levantar os principais problemas neste novo processo de desenvolvimento de *software*. Os problemas levantados focavam-se no processo de desenvolvimento de imagens Docker, no processo de validação dessas imagens, na plataforma que irá distribuir essas imagens, o ambiente Kubernetes que suporta as aplicações conteinerizadas e a forma como as aplicações devem ser configuradas para poderem ser implantadas nesse ambiente de execução de contentores. Para cada um destes problemas foram discutidas soluções. Um dos problemas levantados que fazia parte dos objetivos desta dissertação foi ao nível da elasticidade dos nós do *cluster* Kubernetes. Para isso foi apresentada uma solução com base numa interface que permitia interagir com o componente de orquestração Heat do OpenStack de maneira a tornar os nós do *cluster* Kubernetes elásticos.

Após o caminho necessário para o desenvolvimento de *software* conteinerizado estar concluído fomos explorar e validar através de uma aplicação chamada SmartIoT as soluções sugeridas. Com isso esperava-se demonstrar que esta nova estratégia para o desenvolvimento e execução de *software* trazia várias vantagens face aos mecanismos tradicionais baseados exclusivamente em máquinas virtuais. Inicialmente foram enumerados os requisitos e as razões que levaram a selecionar esta aplicação e depois foram apresentadas as informações gerais sobre a arquitetura deste sistema. Após isso, iniciamos o processo de conteinerização onde esta aplicação foi dividida logicamente por contentores para que fosse possível ter uma visão da quantidade de imagens Docker que teriam de ser criadas, uma por cada um dos elementos da sua arquitetura. Logo após as imagens estarem criadas passamos para a demonstração do *deployment* da aplicação sobre o *cluster* Kubernetes. Tendo a aplicação a correr sobre o ambiente Kubernetes foram realizados um conjunto de avaliações de maneira a testar as principais características e a robustez da plataforma Kubernetes. Com este estudo de caso foi possível concluir os seguintes aspetos:

- As aplicações quando são conteinerizadas e o seu *deployment* é realizado sobre uma plataforma Kubernetes é possível reduzir o tempo de *deployment* em 6 vezes;
- As atualizações ao *software* realizadas num ambiente conteinerizado sobre Kubernetes são mais eficientes, rápidas e simples de serem realizadas;
- As ações de escalabilidade realizadas pelo Kubernetes são mais rápidas e eficientes quando comparadas com os ambientes de máquinas virtuais;
- A utilização dos grupos de escalabilidade do OpenStack permitem tornar o cluster Kubernetes mais elástico aumentando a autonomia e eficiência do mesmo;
- O cluster Kubernetes é uma plataforma robusta que é capaz de lidar e recuperar de vários tipos de falhas, tais como falha de um nó do cluster, falha do processo kubelet, falha da plataforma de conteinerização Docker. Para além disso, esta plataforma é capaz de lidar com falhas ao nível das aplicações que suporta, aumentando a robustez das mesmas;
- A utilização do conceito de CI/CD aliada com as tecnologias de gestão e conteinerização de aplicações permite reduzir a disparidade entre o ambiente de desenvolvimento e de produção, reduzindo o risco de falhas no momento de *deployment* das aplicações no ambiente de produção;
- Devido à arquitetura dos contentores, aos mecanismos de elasticidade e ao processo

automático de atualização de *software* realizado pelo Kubernetes a utilização de recursos é reduzida, quando comparados com os ambientes de máquinas virtuais.

- O desenvolvimento de aplicações conteinerizadas torna-se mais simples quando utilizamos a plataforma Kubernetes para realizar o seu *deployment*, devido a sistemas e objetos que este tem nativamente implementados;
- Combinando os fatores da redução do tempo de *deployment*, das atualizações realizadas ao *software* serem automáticas e rápidas, da redução da disparidade entre o ambiente de desenvolvimento e produção e a facilidade no desenvolvimento de aplicações proporcionado pelo Kubernetes é possível concluir que a conteinerização de aplicações e a sua gestão pela plataforma Kubernetes permite reduzir o tempo global do desenvolvimento de *software*;

Para além dos pontos anteriores, tendo em conta todo o trabalho realizado no âmbito desta dissertação é possível concluir que a conteinerização de aplicações é uma tarefa complexa que requer uma aprendizagem de novos conceitos que influenciam todas as fases do desenvolvimento de *software*. Para além disso, uma organização que queira adotar estas tecnologias tem de ser munida de uma base tecnológica e de uma infraestrutura física capaz de suportar todos os componentes necessários. Por outro lado, é possível concluir que sem adoção deste novo paradigma é difícil para uma organização manter-se no mercado com um grau de competitividade alto, lidar com prazos de entrega de *software* apertados, ser capaz de se adaptar a alterações tecnológicas, de lidar com fluxos inconstantes de pedidos de clientes de uma forma eficiente.

7.1 Objetivos Concluídos

No Capítulo 1 definimos os objetivos para este projeto, ao qual todos estes objetivos foram concluídos. A verificação da conclusão dos objetivos tem por base os seguintes pontos:

- Para a adoção da conteinerização de aplicações num ambiente de desenvolvimento que utiliza as práticas integração e entrega contínua (CI /CD) foram descritos todos os problemas identificados e possíveis soluções tecnológicas;
- Foi apresentado os tipos de *clusters* Kubernetes necessários no quotidiano da organização e as formas como esses *clusters* poderiam ser criados.
- Foi criada a interface que permite lançar grupos elásticos de máquinas virtuais e dessa forma introduzir elasticidade nos nós do *cluster* Kubernetes.
- Foi identificada a aplicação SmartIoT para ser conteinerizada e implantada no *cluster* Kubernetes que serviu de prova de conceito de maneira a avaliar as soluções apresentadas para a conteinerização e execução de aplicações conteinerizadas.
- Com base nos resultados da avaliação do *deployment* e execução da SmartIoT foram apresentadas as vantagens e desvantagens que este novo paradigma traz às organizações que queiram dar o passo para estes novos paradigmas.

7.2 Trabalho Futuro

Com o estudo realizado nesta dissertação foram identificados elementos que previamente não foram identificados como sendo pontos chaves para esta dissertação. Devido a isso espera-se que estes possam ser desenvolvidos num futuro próximo. Esses pontos são:

- Estudar e avaliar as ferramentas que proporcionam uma centralização dos *logs* das aplicações conteinerizadas nestes ambientes distribuídos;
- Estudar os impactos que a conteinerização de aplicações podem trazer ao nível da segurança de maneira a encontrar soluções que possam minimizar esses problemas de segurança, caso existam;
- Desenvolver um sistema de validação de imagens que seja capaz de validar certas regras e com isso manter um controlo das imagens disponíveis nos repositórios internos.
- Avaliar os impactos do *deployment* de aplicações conteinerizadas sobre ambientes *bare-metal*, principalmente ao nível dos sistemas que fornecem persistência de dados.

Apêndice A

Exemplo: Template para Escalabilidade

```
heat_template_version: 2014-10-16

description: Simple template to deploy an autoscaling group

parameters:
  image:
    type: string
    label: Image ID
    description: Image to be used for compute instance

  flavor:
    type: string
    description: Type of instance (flavor) to be used

  private_network:
    type: string
    label: Private Network ID

  public_network:
    type: string
    label: Public Network ID

  max_size:
    type: number
    label: Max size of stack

  min_size:
    type: number
    label: Min size of stack

  scale_interval:
    type: number
    label: Scale Interval
```

```
    description: Amount of time, in seconds, between scaling activities.

  evaluation_period:
    type: number
    label: Evaluation Period
    description: The amount of time, in seconds, that a resource is evaluated
                 considering a given threshold.

  threshold_high:
    type: number
    label: Thresold High
    description: The average MAXIMUM threshold value for the evaluated resource.

  threshold_low:
    type: number
    label: Thresold Low
    description: The average MINIMUM threshold value for the evaluated resource.

resources:
  cluster:
    type: OS::Heat::AutoScalingGroup
    properties:
      resource:
        type: OS::Nova::Server
        properties:
          image: { get_param: image }
          flavor: { get_param: flavor }
          private_network: {get_param: private_network}
          public_network: {get_param: public_network}
          metadata: {"metering.stack": {get_param: "OS::stack_id"},
                     "ASG_SERVER": true}
          user_data: |
            {{.ScriptFile}}
          user_data_format: RAW
      min_size: { get_param: min_size }
      max_size: { get_param: max_size }

  scale_up_policy:
    type: OS::Heat::ScalingPolicy
    properties:
      adjustment_type: change_in_capacity
      auto_scaling_group_id: {get_resource: cluster}
      cooldown: { get_param: scale_interval }
      scaling_adjustment: '1'

  scale_dw_policy:
    type: OS::Heat::ScalingPolicy
    properties:
      adjustment_type: change_in_capacity
      auto_scaling_group_id: {get_resource: cluster}
      cooldown: { get_param: scale_interval }
```

```
      scaling_adjustment: '-1'

  CPUAlarmHigh:
    type: OS::Ceilometer::Alarm
    properties:
      description: Scale-up policy
      meter_name: cpu_util
      statistic: avg
      period: { get_param: evaluation_period }
      evaluation_periods: '1'
      threshold: { get_param: threshold_high }
      alarm_actions: [{get_attr: [scale_up_policy, alarm_url]}]
      matching_metadata: {'metadata.user_metadata.stack':
                          {get_param: "OS::stack_id"}}
      comparison_operator: gt

  CPUAlarmLow:
    type: OS::Ceilometer::Alarm
    properties:
      description: Scale-down policy
      meter_name: cpu_util
      statistic: avg
      period: { get_param: evaluation_period }
      evaluation_periods: '1'
      threshold: { get_param: threshold_low }
      alarm_actions: [{get_attr: [scale_dw_policy, alarm_url]}]
      matching_metadata: {'metadata.user_metadata.stack':
                          {get_param: "OS::stack_id"}}
      comparison_operator: lt


outputs:
  cluster_size:
    description: This is the current size of the auto scaling group.
    value: {get_attr: [cluster, current_size]}
  server_list:
    description: This is a list of server that are part of the group.
    value: {get_attr: [cluster, outputs_list, name]}
  instance_ips:
    description: the IP address of the instance
    value: {get_attr: [cluster, outputs_list, networks,
            {get_param: network}, 0]}
```

Apêndice B

Exemplo: config-server.json

```
{
    "sys": {
        "logs": {
        "channels": {
        "http_listener": {
          "active": false
        },
        "delivery_worker": {
          "active": true
        },
        "processing_worker": {
          "active": true
        },
        "http_listener": {
          "active": true
        },
        "retry_delivery_worker": {
          "active": true
        }
      }
        }
    },
    "services": {
        "catalog": {
            "database": {
                "driver": "org.postgresql.Driver",
                "url": "jdbc:postgresql://smartiot-postgres-svc:5432
                                        /smartiot",
                "user": "catalog",
                "password": "catalog",
                "pool": {
                    "initial_size": 1,
                    "max_size": 1,
                    "max_idle": 1,
                    "min_idle": 1,
                    "max_wait_millis": 1000
```

```
            }
        }
    },
    "history": {
        "database": {
            "driver": "org.postgresql.Driver",
            "url": "jdbc:postgresql://smartiot-postgres-svc:5432
                                    /smartiot",
            "user": "history",
            "password": "history",
            "pool": {
                "initial_size": 1,
                "max_size": 1,
                "max_idle": 1,
                "min_idle": 1,
                "max_wait_millis": 1000
            }
        }
    },
    "tokens_repository": {
        "user": "ignored",
        "password": "ignored",
        "sentinels": "smartiot-redis-sentinel-svc:6379",
        "cluster": "mymaster",
        "cmd_timeout": 1000,
        "token_validity": 3600,
        "pool": {
            "initial_size": 20,
            "max_size": 20,
            "max_idle": 20,
            "min_idle": 20,
            "max_wait_millis": 1000
        }
    },
    "input_queue": {
        "producer": {
            "bootstrap_servers":
                "smartiot-kafka-0.smartiot-kafka-hsvc:9092,
            smartiot-kafka-1.smartiot-kafka-hsvc:9092,
            smartiot-kafka-2.smartiot-kafka-hsvc:9092",
            "connect_timeout": 1000,
            "request_timeout": 100,
            "reconnect_backoff":100,
            "acks": "1",
            "block_on_buffer_full": false,
            "batch_size": 10
        },
        "consumer": {
            "bootstrap_servers":
                "smartiot-kafka-0.smartiot-kafka-hsvc:9092,
            smartiot-kafka-1.smartiot-kafka-hsvc:9092,
```

```
                smartiot-kafka-2.smartiot-kafka-hsvc:9092",
                "timeout": 15000,
                "heartbeat_interval": 2000
            }
        },
        "delivery_queue": {
            "user": "ignored",
            "password": "ignored",
            "sentinels": "smartiot-redis-sentinel-svc:6379",
            "cluster": "mymaster",
            "cmd_timeout": 1000,
            "pool": {
                "initial_size": 20,
                "max_size": 20,
                "max_idle": 20,
                "min_idle": 20,
                "max_wait_millis": 1000
            }
        },
        "delivery_scheduler": {
            "user": "ignored",
            "password": "ignored",
            "sentinels": "smartiot-redis-sentinel-svc:6379",
            "cluster": "mymaster",
            "cmd_timeout": 1000,
            "pool": {
                "initial_size": 10,
                "max_size": 10,
                "max_idle": 10,
                "min_idle": 10,
                "max_wait_millis": 1000
            }
        },
        "processing_worker": {
            "batch_size": 1
        },
        "delivery_worker": {
        },
        "retry_delivery_worker" : {
        },
        "http_listener": {
        }
    },
    "interfaces": {
        "delivery_worker": {
            "num_threads": 10
        },
        "http_listener": {
            "min_threads": "1",
            "max_threads": "10",
            "listening_port": 3000
```

```
            }
        }
}
```

Apêndice C

Script de Instalação do Worker Node Kubernetes

```
#!/bin/sh

# Repository to downaload rpms
REPO="ext-testing"

# Masters IP address
MASTER_1="192.168.14.152"
MASTER_2="192.168.14.153"
MASTER_3="192.168.14.154"

# Kubernetes DNS Address
KUBE_DNS="101.64.0.10"

# Registry to download docker images
IMAGE_REG="repo-docker.example.com"

# OpenStack Configurations
URL_OS="http://10.112.76.130:5000/v2.0"
USER_OS="user-name"
PASSWD_OS="XXXXXX"
TENANT_ID="a0e0712dsomeide255e3ae856"

# CA Certificate (file ca.pem)
CA_CERT=" "

# Private Master Key (file node-key.pem)
NODE_KEY=" "

# Master cerficate (file node.pem)
NODE=" "

# Admin Certificate
ADMIN=" "
```

```
# Admin Key
ADMIN_KEY=" "

#Install packages
yum clean all
yum-config-manager --enable $REPO
yum --enablerepo=$REPO -y install docker
yum --enablerepo=$REPO -y install flannel
yum --enablerepo=$REPO -y install kubernetes-node.x86_64

#UPDATE CONFIGURATION FILES
#File: Kubelet
echo '# kubernetes kubelet (minion) config' > /etc/kubernetes/kubelet
echo 'KUBELET_ADDRESS="--address=0.0.0.0 --port=10250"' >> /etc/kubernetes/kubelet
echo 'KUBELET_POD_INFRA_CONTAINER="--pod-infra-container-image=
        repo-docker.example.com/rhel7-pod-infrastructure"' >> /etc/kubernetes/kubelet
echo 'KUBELET_ARGS="--kubeconfig=/etc/kubernetes/kubeletconfig
        --require-kubeconfig=true
        --pod-manifest-path=/etc/kubernetes/manifests
        --cgroup-driver=systemd
        --cloud-provider=openstack
        --cloud-config=/etc/kubernetes/cloudprovider"' >> /etc/kubernetes/kubelet
echo 'KUBE_LOGTOSTDERR="--logtostderr=true --v=0"' >> /etc/kubernetes/kubelet
echo 'KUBE_ALLOW_PRIV="--allow-privileged=true"' >> /etc/kubernetes/kubelet
echo "KUBELET_DNS_ARGS='--cluster-dns=$KUBE_DNS
        --cluster-domain=kubernetes'" >> /etc/kubernetes/kubelet

#File: kubeletconfig
echo "
apiVersion: v1
clusters:
- name: cluster-ssl
  cluster:
    certificate-authority: /etc/kubernetes/pki/ca.pem
    server: https://$MASTER_1
- name: cluster-sslskip
  cluster:
    server: https://$MASTER_1
    insecure-skip-tls-verify: true
contexts:
- context:
    cluster: cluster-ssl
    user: kubelet-devmaster
  name: kubelet-devmaster@cluster-ssl
- context:
    cluster: cluster-sslskip
    user: admin_auth
  name: admin@cluster-sslskip
current-context: admin@cluster-sslskip
kind: Config
```

```
users:
- name: kubelet-devmaster
  user:
    client-certificate: /etc/kubernetes/pki/node.pem
    client-key: /etc/kubernetes/pki/node-key.pem
- name: admin_auth
  user:
   username: admin
   password: AdminPass
" > /etc/kubernetes/kubeletconfig

#File: kubeadminconfig
echo "apiVersion: v1
clusters:
- name: cluster-sslskip
  cluster:
    server: https://$MASTER_1
    insecure-skip-tls-verify: true
- name: cluster-ssl
  cluster:
    certificate-authority: /etc/kubernetes/pki/ca.pem
    server: https://$MASTER_1
contexts:
- context:
    cluster: cluster-ssl
    user: admin_ssl
  name: admin@cluster-ssl
- context:
    cluster: cluster-sslskip
    user: admin_auth
  name: admin@cluster-sslskip
current-context: admin@cluster-ssl
kind: Config
users:
- name: admin_ssl
  user:
    client-certificate: /etc/kubernetes/pki/admin.pem
    client-key: /etc/kubernetes/pki/admin-key.pem
- name: admin_auth
  user:
   username: admin
   password: AdminPass
" > /etc/kubernetes/kubeadminconfig

#File: cloudprovider
echo "
[Global]
auth-url=$URL_OS http://10.112.76.130:5000/v2.0
username=$USER_OS ci-ctidois
password=$PASSWD_OS c1-ctidois
region=RegionOne
```

```
tenant-id=$TENANT_ID a0e0712d144648a58f00ce215e8ae856
" > /etc/kubernetes/cloudprovider

#File: /etc/sysconfig/flannel
echo "FLANNEL_OPTIONS='-etcd-endpoints=http://$MASTER_1:2379,
        http://$MASTER_2:2379,http://$MASTER_3:2379
        -etcd-prefix /atomic.io/network'" > /etc/sysconfig/flanneld

#Configure Registry
echo "INSECURE_REGISTRY='--insecure-registry=$IMAGE_REG'" >> /etc/sysconfig/docker

#Create /etc/kubernetes/manifests dir
mkdir /etc/kubernetes/manifests

#Keys
mkdir /etc/kubernetes/pki
echo $NODE_KEY > /etc/kubernetes/pki/node-key.pem
echo $NODE > /etc/kubernetes/pki/node.pem
echo $CA_CERT > /etc/kubernetes/pki/ca.pem
echo $ADMIN > /etc/kubernetes/pki/admin.pem
echo $ADMIN_KEY > /etc/kubernetes/pki/admin-key.pem

#Restart and Enable Services
systemctl enable kubelet
systemctl restart kubelet
systemctl enable flanneld
systemctl restart flanneld
systemctl enable docker
systemctl restart docker
systemctl enable kube-proxy
systemctl restart kube-proxy
```

Apêndice D

Ficheiro haproxy.cfg

```
#---------------------------------------------------------------------
# Global settings
#---------------------------------------------------------------------
global
    log 127.0.0.1 local0
    log 127.0.0.1 local1 notice

#---------------------------------------------------------------------
# common defaults that all the 'listen' and 'backend' sections will
# use if not designated in their block
#---------------------------------------------------------------------
defaults
    mode                    http
    log                     global
    option                  dontlognull
    timeout connect         5000ms
    timeout client          50000ms
    timeout server          50000ms

#---------------------------------------------------------------------
#HAProxy Monitoring Config
#---------------------------------------------------------------------
listen stats
    bind *:8888                #Haproxy Monitoring run on port 8888
    mode http
    balance
    stats uri /stats
    stats auth admin:123qwe
    stats admin if TRUE

#---------------------------------------------------------------------
# main frontend which proxys to the backends
#---------------------------------------------------------------------
frontend  kube_api_req
        mode tcp
        bind *:443
```

```
        use_backend        kube_api

#---------------------------------------------------------------------
# balancing between the various backends
#---------------------------------------------------------------------
backend kube_api
    balance     roundrobin
    server  kube-node1 10.112.76.180:5443 check
    server  kube-node2 10.112.76.181:5443 check
    server  kube-node3 10.112.76.183:5443 check
```

Apêndice E

Teste: Variação do Número de Réplicas

Nesta avaliação pretende-se saber a quantidade de pedidos respondidos pela variação do número de instâncias dedicado a um serviço. Para cada um dos testes foi realizado três medições de maneira a obter a média dos pedidos respondidos. Com isto pretende-se reduzir os efeitos da variação do estado da rede no momento da avaliação.

Nesta prova foram realizadas 4 experiências uma para cada grupo de instâncias. A primeira experiência tinha apenas uma réplica do Protocol Listener, a segunda experiência continha 2 réplicas do Protocol Listener e assim sucessivamente até chegarmos a 4 réplicas. Neste cenário o Vegeta foi configurado para uma taxa de 600 pedidos por segundo durante 10 segundos. Os valores recolhidos foram a quantidade de pedidos que a aplicação SmartIoT conseguiu responder durante os 10 segundos. Os resultados obtidos para a quantidade total de pedidos processado são apresentados na Tabela E.1.

Tabela E.1: Pedidos em função da variação do número de réplicas do Protocol Listener.

N.º Protocol Listeners	1	2	3	4
Quantidade Total de Pedidos	2372	3479	4524	5373
Taxa de Resposta (pedidos/segundo)	237.2	347.9	452.4	537.3

Bibliografia

[1] Michael Cusumano. Cloud computing and saas as new computing platforms. *Communications of the ACM*, 53(4):27–29, 2010.

[2] Docker: Official website. https://www.docker.com/. Accessed: 2016-12-20.

[3] Kubernetes: Official website. http://kubernetes.io/. Accessed: 2016-12-19.

[4] Altice Labs: Official website. http://www.alticelabs.com/pt/. Accessed: 2016-09-26.

[5] Yashpalsinh Jadeja and Kirit Modi. Cloud computing-concepts, architecture and challenges. In *Computing, Electronics and Electrical Technologies (ICCEET), 2012 International Conference on*, pages 877–880. IEEE, 2012.

[6] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. 2009.

[7] Peter Mell and Tim Grance. The nist definition of cloud computing. 2011.

[8] Amazon EC2. https://aws.amazon.com/pt/ec2/. Accessed: 2016-12-22.

[9] Microsoft Azure. https://azure.microsoft.com/pt-pt/. Accessed: 2016-12-22.

[10] Google app engine. https://cloud.google.com/appengine/. Accessed: 2016-12-22.

[11] Heroku. https://www.heroku.com/. Accessed: 2016-12-22.

[12] Google docs. https://docs.google.com/. Accessed: 2016-12-22.

[13] Google gmail. www.gmail.com. Accessed: 2016-12-22.

[14] OpenStack: official page. http://www.openstack.org/. Accessed: 2016-10-12.

[15] Apache Cloud Stack: open source cloud computing. https://cloudstack.apache.org/index.html. Accessed: 2016-12-04.

[16] Nancy Jain and Sakshi Choudhary. Overview of virtualization in cloud computing. In *Colossal Data Analysis and Networking (CDAN), Symposium on*, pages 1–4. IEEE, 2016.

[17] Michael Fenn, Michael A Murphy, Jim Martin, and Sebastien Goasguen. An evaluation of kvm for use in cloud computing. In *Proc. 2nd International Conference on the Virtual Computing Initiative, RTP, NC, USA*, 2008.

[18] Virtzone: The difference between a 'type 1' hypervisor and a 'type 2' hypervisor. www.virtzone.net/the-difference-between-a-type-2-hypervisor-and-a-type-1-hypervisor/. Accessed: 2016-11-29.

[19] P Vijaya Vardhan Reddy and Lakshmi Rajamani. Evaluation of different hypervisors performance in the private cloud with sigar framework. *International Journal of Advanced Computer Science and Applications*, 5(2), 2014.

[20] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and YC Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*, page 1. ACM, 2016.

[21] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[22] Jeff Nickoloff. *Docker in Action.* Manning Publications Co., 2016.

[23] Linux Man Pages: cgroups - linux control groups. http://man7.org/linux/man-pages/man7/cgroups.7.html, . Accessed: 2016-12-01.

[24] Linux Man Pages: namespaces - overview of linux namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html, . Accessed: 2016-12-01.

[25] Linux Man Pages: chroot - change root directory. http://man7.org/linux/man-pages/man2/chroot.2.html, . Accessed: 2017-08-19.

[26] Docker: What is a container. https://www.docker.com/what-container. Accessed: 2017-08-19.

[27] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Y. C. Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, pages 1:1–1:13, New York, NY, USA, 2016. ACM. ISBN: 978-1-4503-4300-8. doi:10.1145/2988336.2988337.

[28] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[29] Thanh Bui. Analysis of docker security. *arXiv preprint arXiv:1501.02967*, 2015.

[30] Miguel G Xavier, Marcelo V Neves, Fabio D Rossi, Tiago C Ferreto, Timoteo Lange, and Cesar AF De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240. IEEE, 2013.

[31] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, March 2007. ISSN: 0163-5980. doi:10.1145/1272998.1273025.

[32] Chenxi Wang. InfoWorld: Containers 101: Linux containers and docker explained. www.infoworld.com/article/3072929/linux/containers-101-linux-containers-and-docker-explained.html. Accessed: 2016-12-01.

[33] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. Virtualization vs containerization to support paas. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 610–614. IEEE, 2014.

[34] James Lewis and Martin Fowler. Microservices. *http://martinfowler. com/articles/microservices. html. Acessado em*, 1(03):2016, 2014.

[35] Dmitry Namiot and Manfred Sneps-Sneppe. On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27, 2014.

[36] Nginx: "the future of application development and delivery is now". https://www.nginx.com/resources/library/app-dev-survey/. Accessed: 2017-08-22.

[37] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 23–27, 2013.

[38] Guilherme Galante and Luis Carlos E de Bona. A survey on cloud computing elasticity. In *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*, pages 263–270. IEEE, 2012.

[39] Chuanqi Kan. Docloud: An elastic cloud platform for web applications based on docker. In *2016 18th International Conference on Advanced Communication Technology (ICACT)*, pages 478–483. IEEE, 2016.

[40] Wikipedia: The free encyclopedia. https://en.wikipedia.org/wiki/Waterfall_model. Accessed: 2016-12-02.

[41] Jez Humble and Joanne Molesky. Why enterprises must adopt devops to enable continuous delivery. *Cutter IT Journal*, 24(8):6, 2011.

[42] F Oliveira, T Eilam, P Nagpurkar, C Isci, M Kalantar, W Segmuller, and E Snible. Delivering software with agility and quality in a cloud environment. *IBM Journal of Research and Development*, 60(2-3):10–1, 2016.

[43] Suzie Prince. mind the PRODUCT: The product managers' guide to continuous delivery and devops. http://www.mindtheproduct.com/2016/02/what-the-hell-are-ci-cd-and-devops-a-cheatsheet-for-the-rest-of-us/. Accessed: 2016-12-2.

[44] Martin Fowler and Matthew Foemmel. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, page 122, 2006.

[45] G2 Crowd: Best version control systems. https://www.g2crowd.com/categories/version-control-systems. Accessed: 2016-11-6.

[46] Jenkins: Official website. https://jenkins.io/. Accessed: 2017-08-18.

[47] John Ferguson Smart. *Jenkins: The Definitive Guide: Continuous Integration for the Masses*. "O'Reilly Media, Inc.", 2011.

[48] Lianping Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32 (2):50–54, 2015.

[49] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[50] OpenStack: Companies supporting the openstack foundation. http://www.openstack.org/foundation/companies/, . Accessed: 2016-10-12.

[51] The OpenStack Blog: Introducing openstack. http://www.openstack.org/blog/2010/07/introducing-openstack/, . Accessed: 2016-10-12.

[52] OpenStack Documentation: Manage flavors. https://docs.openstack.org/horizon/latest/admin/manage-flavors.html. Accessed: 2017-09-24.

[53] Omar Khedher. *Mastering OpenStack*. Packt Publishing Ltd, 2015.

[54] Dan Radez. *OpenStack Essentials*. Packt Publishing Ltd, 2015.

[55] OpenStack Documentation: Heat orchestration template (hot) specification. https://docs.openstack.org/heat/latest/template_guide/hot_spec.html. Accessed: 2017-09-23.

[56] OpenStack Documentation: Openstack resource types. https://docs.openstack.org/heat/latest/template_guide/openstack.html. Accessed: 2017-08-04.

[57] Google: Cloud platform. https://cloud.google.com/. Accessed: 2017-05-05.

[58] Google gcloud: Overview. https://cloud.google.com/sdk/gcloud/. Accessed: 2017-05-05.

[59] Google Cloud SDK: Documentation. https://cloud.google.com/sdk/docs/.

[60] Google gcloud: Reference. https://cloud.google.com/sdk/gcloud/reference/. Accessed: 2017-05-05.

[61] Google: Compute engine. https://cloud.google.com/compute/. Accessed: 2017-05-05.

[62] Google gcloud: compute instance-groups. https://cloud.google.com/sdk/gcloud/reference/compute/instance-groups/, . Accessed: 2017-05-05.

[63] Google gcloud: compute instance-groups managed set-autoscaling. https://cloud.google.com/sdk/gcloud/reference/compute/instance-groups/managed/set-autoscaling, . Accessed: 2017-05-05.

[64] Amazon: Ec2. https://aws.amazon.com/pt/ec2/, . Accessed: 2017-05-05.

[65] Amazon: Aws command line interface documentation. https://aws.amazon.com/pt/documentation/cli/, . Accessed: 2017-05-05.

[66] Amazon: Aws autoscaling. http://docs.aws.amazon.com/cli/latest/reference/autoscaling/index.html, . Accessed: 2017-05-05.

[67] Amazon: Aws create-auto-scaling-group. http://docs.aws.amazon.com/cli/latest/reference/autoscaling/create-auto-scaling-group.html, . Accessed: 2017-05-05.

[68] Go: Official website. https://golang.org/. Accessed: 2017-04-18.

[69] Gophercloud: Official website. http://www.gophercloud.io/. Accessed: 2017-04-18.

[70] Rackspace: Official website. https://www.rackspace.com. Accessed: 2017-04-18.

[71] Cobra: Official github page. https://github.com/spf13/cobra. Accessed: 2017-04-18.

[72] Docker Guides: Docker overview. https://docs.docker.com/engine/docker-overview/, . Accessed: 2017-08-06.

[73] Docker Reference: Dockerfile reference. https://docs.docker.com/engine/reference/builder/, . Accessed: 2017-08-06.

[74] James Turnbull. *The docker book*. Lulu. com, 2014.

[75] Docker Guides: About images, containers, and storage drivers. https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/. Accessed: 2017-08-06.

[76] Docker Hub: Official website. https://hub.docker.com/, . Accessed: 2017-07-31.

[77] Docker Product Manuals: Docker registry. https://docs.docker.com/registry/, . Accessed: 2017-07-31.

[78] Apache Mesos: Official website. http://mesos.apache.org/, . Accessed: 2016-12-19.

[79] Mesos: Architecture. http://mesos.apache.org/documentation/latest/architecture/, . Accessed: 2017-09-21.

[80] Mesosphere: Marathon. https://mesosphere.github.io/marathon/, . Accessed: 2016-12-19.

[81] Mesos: Features. https://mesosphere.github.io/marathon/#features. Accessed: 2017-09-21.

[82] Marathon: Stateful applications using external persistent volumes. https://mesosphere.github.io/marathon/docs/external-volumes.html. Accessed: 2017-09-25.

[83] HAProxy: The reliable, high performance tcp/http load balancer. www.haproxy.org/. Accessed: 2016-12-2.

[84] René Peinl, Florian Holzschuher, and Florian Pfitzer. Docker cluster management for the cloud-survey results and own solution. *Journal of Grid Computing*, 14(2):265–282, 2016.

[85] Docker: Docker swarm. https://www.docker.com/products/docker-swarm. Accessed: 2016-12-20.

[86] Michael Hausenblas. *Docker Networking and Service Discovery*. O'Reilly Media, 2016.

[87] Docker Swarm: Feature highlights. https://docs.docker.com/engine/swarm/#feature-highlights, . Accessed: 2017-09-20.

[88] Docker Compose: Overview. https://docs.docker.com/compose/overview/, . Accessed: 2017-09-22.

[89] RedMonk: Kubernetes revisited, top corporate contributors over time. http://redmonk.com/fryan/2016/03/21/kubernetes-revisited-top-corporate-contributors-over-time/. Accessed: 2017-08-07.

[90] Cloud Native Computing Foundation: Projects. https://www.cncf.io/projects/. Accessed: 2017-09-21.

[91] Kubernetes: What is kubernetes? http://kubernetes.io/docs/whatisk8s/. Accessed: 2016-12-18.

[92] Kubernetes Concepts: Pod overview. https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/, . Accessed: 2017-08-07.

[93] Kubernetes Concepts: Pod lifecycle. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/, . Accessed: 2017-08-07.

[94] Kubernetes Concepts: Volumes. https://kubernetes.io/docs/concepts/storage/volumes/, . Accessed: 2017-08-07.

[95] Kubernetes Concepts: Services. https://kubernetes.io/docs/concepts/services-networking/service/, . Accessed: 2017-08-07.

[96] Kubernetes Concepts: Labels and selectors. https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/, . Accessed: 2017-08-07.

[97] Kubernetes Concepts: Replica sets. https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/, . Accessed: 2017-08-17.

[98] Kubernetes Concepts: Deployments. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/, . Accessed: 2017-08-17.

[99] Kubernetes Concepts: Statefulsets. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/, . Accessed: 2017-08-17.

[100] Kubernetes Concepts: Headless services. https://kubernetes.io/docs/concepts/services-networking/service/#headless-services, . Accessed: 2017-08-26.

[101] Kubernetes Documentation: Dns service. https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/, . Accessed: 2017-07-31.

[102] Kubernetes Tasks: Configure containers using a configmap. https://kubernetes.io/docs/tasks/configure-pod-container/configmap/, . Accessed: 2017-08-23.

[103] Kubernetes Concepts: Secrets. https://kubernetes.io/docs/concepts/configuration/secret/, . Accessed: 2017-08-23.

[104] Kubernetes Concepts: Persistent volumes. https://kubernetes.io/docs/concepts/storage/persistent-volumes/, . Accessed: 2017-08-23.

[105] Kubernetes Documentation: Performing a rolling update. https://kubernetes.io/docs/tutorials/kubernetes-basics/update-intro/, . Accessed: 2017-07-28.

[106] X-Team: Introduction to kubernetes architecture. https://x-team.com/blog/introduction-kubernetes-architecture/, . Accessed: 2017-08-17.

[107] CoreOS: etcd. https://coreos.com/etcd. Accessed: 2017-08-16.

[108] Kubernetes Reference: kube-apiserver. https://kubernetes.io/docs/admin/kube-apiserver/, . Accessed: 2017-08-16.

[109] Kubernetes Reference: Overview of kubectl. https://kubernetes.io/docs/user-guide/kubectl-overview/, . Accessed: 2017-08-26.

[110] Kubernetes Concepts: Cluster networking. https://kubernetes.io/docs/concepts/cluster-administration/networking/, . Accessed: 2017-08-26.

[111] Github: Flannel. https://github.com/coreos/flannel, . Accessed: 2017-08-26.

[112] Docker Documentation: docker commit. https://docs.docker.com/engine/reference/commandline/commit/. Accessed: 2017-07-31.

[113] Rui Shu, Xiaohui Gu, and William Enck. A study of security vulnerabilities on docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 269–280. ACM, 2017.

[114] Docker: Create a base image. https://docs.docker.com/engine/userguide/eng-image/baseimages/. Accessed: 2017-07-24.

[115] Moby/ mkimage-yum.sh. https://github.com/moby/moby/blob/master/contrib/mkimage-yum.sh. Accessed: 2017-07-11.

[116] Alpine: docker-alpine-3.6. https://github.com/gliderlabs/docker-alpine/blob/c6cb2bbffbc92d4c35cdaf584dd1c43a860104ea/versions/library-3.6/Dockerfile. Accessed: 2017-07-24.

[117] JFrog: Artifactory. https://www.jfrog.com/artifactory/. Accessed: 2017-07-31.

[118] Google Trends: Comparação da popularidade das plataformas kubernetes, docker swarm e mesos. https://trends.google.pt/trends/explore?date=today%205-y&q=Kubernetes,Docker%20Swarm,Mesos. Accessed: 2017-09-22.

[119] Kubernetes: Creating a custom cluster from scratch. https://kubernetes.io/docs/getting-started-guides/scratch/. Accessed: 2017-07-30.

[120] Ansible: Official website. https://www.ansible.com/. Accessed: 2017-07-30.

[121] Kubernetes: Installing kubernetes on-premise/cloud providers with kubespray. https://kubernetes.io/docs/getting-started-guides/kubespray/. Accessed: 2017-07-30.

[122] Kubernetes Decomentation: Fedora via ansible. https://kubernetes.io/docs/getting-started-guides/fedora/fedora_ansible_config/, . Accessed: 2017-07-30.

[123] Projeto GitHub: Kubernetes ansible. https://github.com/kubernetes/contrib/tree/master/ansible, . Accessed: 2017-07-30.

[124] GitHub: Ha-kubernetes-ansible. https://github.com/pawankkamboj/HA-kubernetes-ansible. Accessed: 2017-07-30.

[125] Kubernetes Documentation: Running kubernetes locally via minikube. https://kubernetes.io/docs/getting-started-guides/minikube/. Accessed: 2017-07-30.

[126] Kubernetes Documentation: Install minikube. https://kubernetes.io/docs/tasks/tools/install-minikube/. Accessed: 2017-07-30.

[127] Kubernetes Documentation: Operating etcd clusters for kubernetes. https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/. Accessed: 2017-07-28.

[128] Algoritmo de Raft. https://raft.github.io/. Accessed: 2017-07-28.

[129] CoreOS ETCD: Frequently asked questions. https://github.com/coreos/etcd/blob/master/Documentation/faq.md. Accessed: 2017-07-28.

[130] Kubernetes Setup: Building high-availability clusters. https://kubernetes.io/docs/admin/high-availability/. Accessed: 2017-08-23.

[131] Google Cloud Plataform: Adicionar balanceamento de carga. https://cloud.google.com/compute/docs/load-balancing/. Accessed: 2017-07-30.

[132] Keepalived: Official website. http://www.keepalived.org/. Accessed: 2017-07-30.

[133] OpenStack Heat Documentation: Os::heat::autoscalinggroup. https://docs.openstack.org/heat/latest/template_guide/openstack.html#OS::Heat::AutoScalingGroup, . Accessed: 2017-07-26.

[134] OpenStack Heat Documentation: Os::nova::server. https://docs.openstack.org/heat/latest/template_guide/openstack.html#OS::Nova::Server, . Accessed: 2017-07-26.

[135] Go Lang: Package template. https://golang.org/pkg/text/template/, . Accessed: 2017-07-26.

[136] Go Lang: Package parse. https://golang.org/pkg/text/template/parse/, . Accessed: 2017-07-26.

[137] OpenStack Ceilometer Documentation: Telemetry sample configuration files. https://docs.openstack.org/mitaka/config-reference/telemetry/sample-configuration-files.html. Accessed: 2017-07-27.

[138] Red Hat: Configuring instances at boot time. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux_OpenStack_Platform/4/html/End_User_Guide/user-data.html. Accessed: 2017-07-27.

[139] Apache Kafka: Official website. https://kafka.apache.org/. Accessed: 2017-07-10.

[140] Apache Zookeeper: Official website. http://zookeeper.apache.org/. Accessed: 2017-07-10.

[141] Redis: Official website. https://redis.io/. Accessed: 2017-07-10.

[142] PostgreSQL: Official website. https://www.postgresql.org/. Accessed: 2017-07-10.

[143] Moby Project: Official website. https://mobyproject.org/. Accessed: 2017-07-11.

[144] Kubernetes Setup installing docker. https://kubernetes.io/docs/setup/independent/install-kubeadm/#installing-docker. Accessed: 2017-08-24.

[145] reliable, scalable redis on kubernetes. https://github.com/kubernetes/examples/tree/master/staging/storage/redis, . Accessed: 2017-08-02.

[146] redis sentinel documentation. https://redis.io/topics/sentinel, . Accessed: 2017-08-02.

[147] Kubernetes Concepts replication controller. https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/. Accessed: 2017-08-02.

[148] Vegeta github. https://github.com/tsenart/vegeta. Accessed: 2017-09-14.